AD-A052 726 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
AUTOMATED COMPILER TEST CASE GENERATION.(U)
FEB 78 P T BERNING, E R ANDERSON, F C BELZ F30602-76-C-0255
RADC-TR-78-30 NL

UNCLASSIFIED

1 OF 2
AD
A052 726

AD A052726

RADC-TR-78-30
Final Technical Report
February 1978

AUTOMATED COMPILER TEST CASE GENERATION

Paul T. Berning
Eric R. Anderson
Frank C. Belz

TRW Defense and Space Systems Group

AD NO.
DDC FILE COPY

Approved for public release; distribution unlimited.

DDC
APR 17 1978
F

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-30 has been reviewed and is approved for publication.

APPROVED: Robert E. Stover, Jr.

ROBERT E. STOVER, JR.
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

18 RADC 19 TR-78-30

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
RADC-TR-78-30

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
6 AUTOMATED COMPILER TEST CASE GENERATION.

5. TYPE OF REPORT & PERIOD COVERED
9 Final Technical Report.
Apr[redacted]976 - Dec[redacted]977

6. PERFORMING ORG. REPORT NUMBER
N/A

7. AUTHOR(s)
10 Paul T. Berning,
Eric R. Anderson
Frank C. Belz

8. CONTRACT OR GRANT NUMBER(s)
15 F30602-76-C-0255

9. PERFORMING ORGANIZATION NAME AND ADDRESS
TRW Defense and Space Systems Group
One Space Park
Redondo Beach CA 90278

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
P.E. 62702F
J.O. 55811218

11. CONTROLLING OFFICE NAME AND ADDRESS
Rome Air Development Center (ISIS)
Griffiss AFB NY 13441

12. REPORT DATE
11 Feb[redacted]978

13. NUMBER OF PAGES
103 12 111p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Same 16 5581 17 12

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Robert E. Stover (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Compiler	Automated
Test Case	Synthesizer
Validation	Analyzer
Verification	Generation
SEMANOL	Specification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report discusses the overall design of a software tool for the automation of validation of compilers for conformance to the specification of the high-order programming language they process. Such compiler validation is currently tedious, time-consuming, error-prone, and not completely effective. Improvements in methods of compiler testing have long been sought by the Air Force and other DoD agencies.

The generation of test cases for compiler validation is here envisioned as

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 637 JOB

a two-step process. The starting point for this implementation is the SEMANOL tool, which consists of a machine-representable exact specification of a high-order language that is used to check the consistency of the HOL specification and to directly execute programs written in that HOL. SEMANOL is used by the Analyzer to generate constraints to which compiler test cases must adhere. The Synthesizer then uses these constraints to generate the test cases via a tree-building process.

Although a considerable degree of human intervention is still required in the development of compiler test programs when a tool such as the one designed under this effort is employed, a vast improvement in the process of compiler validation and verification is expected to result.

This report concludes with a sizeable bibliography covering the areas of artifical intelligence and validation and verification of software.

Abstract

This research study sought to find methods by which test cases for a compiler could be generated automatically. It based its investigation upon the use of SEMANOL, a formal notation for programming language specification, as the means by which the syntax and semantics of the language (for which compiler test cases are to be generated) are described. A SEMANOL specification is the assumed primary input to this automated process.

This research established a design framework in which implementation, incremental improvement, and experimentation can be conducted. Two programs, an Analyzer and Synthesizer, were designed at a high level of abstraction, and a variety of heuristic options were formulated and evaluated. The net result is that a basis for prototype implementation of a usefully automated compiler test case generation system was developed; this design basis forms the body of this report.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
SPECIAL
A

CONTENTS

1. INTRODUCTION 1

2. THE SEMANOL SYSTEM 5
   2.1 The SEMANOL Metalanguage 5
   2.2 The Structure of a SEMANOL Specification 8
   2.3 The SEMANOL Interpreter 16

3. THE GENERAL APPROACH 19
   3.1 A Basis For Testing 19
   3.2 The TRW Approach To Test Generation 22
   3.3 Observations On This Approach 31

4. A TEST GENERATOR DESIGN 34
   4.1 The Synthesizer 34
   4.2 The Analyzer 67
   4.3 Test Effectiveness Measurement 89

5. CONCLUSIONS 92

6. REFERENCES 93

7. BIBLIOGRAPHY 95

EVALUATION

The effort entitled "Automated Compiler Test Case Generation" was undertaken because the increased use of high-order programming languages for computer programming within the Air Force has resulted in concern for the accuracy and reliability of processing with such languages. Of particular concern is the compliance of the compiler, the software tool which translates programs written in a HOL into machine language, with the specification of the HOL it is intended to process. The failure of compilers to properly process HOLs has an adverse and possibly disastrous effect upon the development and maintenance of Air Force weapons systems software, as this software may not produce the expected results upon execution as a consequence.

Software packages for compiler validation currently exist and have been implemented for such high-order languages as FORTRAN, COBOL, JOVIAL (J3), and JOVIAL (J73/I). These validation systems consist of a series of modules or programs to successively test the various features of the language under consideration, and certain combinations of these features. They have achieved significant results in the testing of operational compilers within the Air Force. However, they suffer from the drawbacks of being tedious and time-consuming to write and execute, error-prone, and not completely effective.

This effort was undertaken under TPO 5, C^3 Availability, and TRW Space and Defense Systems Group has employed the RADC developed SEMANOL tool, which exactly and completely specifies any high-order language as a base from which to develop a method for the automation of compiler test case generation. This production of test programs is a two-step process. First, the Analyzer uses the SEMANOL specification of the given language to generate a system of constraints which the compiler test programs must satisfy. The Synthesizer then applies these constraints to a tree-building process, testing at each mode for satisfaction of the constraints and only continuing along paths for which they are satisfied, to generate the test cases themselves.

Although a considerable degree of human intervention is still required to develop the compiler test programs, it is expected that this concept can be implemented to some extent in operational situations, to expedite the process of compiler validation. This effort can thus be viewed as a first step toward total automation of the process of generating more effective compiler test programs.

Robert E. Stover, Jr

ROBERT E. STOVER, JR.
Project Engineer

1. INTRODUCTION

The acceptance and field installation of compilers that perform unsatisfactorily is distressingly commonplace. Because compilers are fundamental to the development of most software, the effects of attempting to use a poor compiler are widely distributed and especially damaging. Unrealized, and thus undocumented, errors are detected only when people attempt to use the compiler in production program development. Finding errors in this way is a difficult and frustrating experience for the compiler user; it makes people angry and often lessens personal motivation. It also causes project delays because of wasted time in finding and working around such errors, and may even cause major delays if the errors are such that program redesign is required to overcome the compiler deficiency. The difficulties of attempting to work with a poor compiler can hardly be overstated.

It is the role of testing to determine how well a compiler implementation actually does what its specifications say it should do. The quality of a compiler can thereby be known prior to its acceptance and installation. Assertion proving techniques seek to accomplish a similar goal, and to do so in a more comprehensive manner, but have not yet advanced to the point that they can adequately deal with compiler programs. And even if proof techniques could deal with compilers, proof techniques deal with the source language text of the compiler and so must assume (1) that the source text of the compiler is itself translated to machine code correctly and (2) that the host computer and operating system work in a given way. Testing rests on no such intermediate assumptions, as it deals with the compiler and its generated code just as the user actually sees it. Testing thus complements assertion proving techniques by providing assurance that the actual executing form of the compiler, its host machine, and supporting system software perform as was assumed in the proof process. Since testing empirically measures the end-product results of compilation, it is a mandatory element in any prudent method of compiler evaluation and will remain so in the future. The matter of compiler testing thus warrants great attention.

Unfortunately, existing test procedures for compilers have not been as good as one would want because the test programs used are so often inadequate. Even the Jovial Compiler Validation System (JCVS), a superior example of a test set, is uneven and far from complete in its coverage. The problems with current test set construction methods are that they are:

1. Not systematic. That is, there is no formal method of construction which is applied. The tests reflect an ad hoc analysis of the language, the test designer's experience with other compilers (often in other circumstances), individual energy and insight, etc. Even two competent organizations, given the assignment of implementing a test set for a given compiler, would be expected to deliver somewhat different products.

2. Not designed with reference to measures of test effectiveness. This is hardly surprising since generally accepted measures of compiler testing effectiveness do not exist. However, the lack of measurable, clearly defined, objectives means that test construction is conducted in a fuzzy manner without guidance as to what should be tested nor how thorough tests should be.

3. Prepared manually. Very little automated test generation is done at present, although syntax checking test cases can be produced by computer ([Hanford 1970]). Manual test generation compounds the problems already cited by introducing the unevenness of individual performance. Test cases are like other programs, so must be carefully prepared from test specifications. The resulting output quality becomes highly dependent upon the test case implementor. Since test construction is not generally held in high regard by practicing programmers, it is difficult to have this type of work done well.

4. Expensive. As programming is an expensive service, manual test case construction is often not done thoroughly because of the high cost that would be involved in attempting to do so.

This brief discussion suggests the extreme difficulty which now faces anyone who would produce a test set capable of adequately testing a compiler. While a lengthy period of repeated test set improvement can lead to the development of a comprehensive testing procedure, such test improvement is difficult and costly work that is seldom done.

Automation of the test generation process, if feasible, offered the promise of easing the problems cited and so of leading to better testing methods. Thus this project was undertaken as a research study seeking to find algorithms useful for the automatic generation of programs that would constitute an effective test of compilers for a given programming language. While it was never expected that a totally automated test generation procedure could be found, it was believed that enough would be learned so that a promising design approach could be developed for a usefully automated system. TRW's SEMANOL system of formal semantic description was accepted as the basis upon which to develop an automatic test generation system. A SEMANOL specification can be read and "understood" by a computer program; thus it was believed that a computer program might be written to generate semantically significant test cases from this formal specification. It was recognized that changes to the SEMANOL metalanguage, its implementing Interpreter program, and conventions in its use might be required to support test case generation. It was further intended that test effectiveness measures based on SEMANOL specification coverage be developed, and that methods of recording effectiveness would be designed for later inclusion in the Interpreter. The results of this study were to be reported upon in a Final Technical Report, as has been done here, and in an oral presentation at Rome Air Development Center.

TRW conducted this project in a straightforward manner. The technical literature was widely and carefully read, as shown by the substantial bibliography that is part of this report. Although the specific problem under study here has not been considered in the past, a good deal of research in Artificial Intelligence (AI) has some application; this was especially true of work in automatic programming and test data-set generation, but it was also true of other research in problem solving methodolgy and search

techniques. A series of formal SEMANOL specifications of sample languages, largely extracted from Minimal BASIC, was developed for use in (1) manually testing possible algorithms and (2) deriving insights into how search limiting heuristics might be developed. This use of concrete specifications helped keep the research from becoming mired down through focusing upon an overly generalized problem. As with so many other AI problems, this research has largely dealt with seeking ways in which search methods can be made computationally reasonable. We believe we have been somewhat successful in this regard. Test effectiveness measures received limited attention since (1) the measures proposed for conventional application programs also appear appropriate for SEMANOL metaprograms and (2) the problems of automated test case generation relative to even a simple test effectiveness measure could not be fully solved. The implementation of measurement techniques was investigated and the design of Interpreter changes to support measurement was performed; implementation should not be a problem.

The oral presentation given at RADC met its purpose by explaining the test generation problem and describing how the methods developed in this project might lead to a partly automated system. We believe the major contribution made in this research has been to develop a framework in which implementation can reasonably be done, and in which support of a continuing process of refinement can be conducted.

The remainder of this report describes the general design that was developed in performance of this contract, discusses several factors of special importance, and presents specific algorithms for parts of the overall design. A brief description of SEMANOL is included in order that this report be self-sufficient.

2. THE SEMANOL SYSTEM

Since TRW's approach to compiler test case generation is based upon the use of SEMANOL, some knowledge of SEMANOL is needed if the remainder of this report is to be understood. This section is meant to provide that needed background.

A formal SEMANOL specification of a programming language is a program; a program for processing a source language program text written in the programming language being defined. The algorithm expressed by the SEMANOL metaprogram describes a way in which the intended effect of executing any program in the defined language can be realized. That is, the algorithm is an interpretive definition of semantics or, alternatively viewed, the metaprogram describes an interpreter for the defined programming language. SEMANOL thereby provides an operational form of programming language specification.

2.1 The SEMANOL Metalanguage

One sense of the nature of the SEMANOL metalanguage used in writing these operational specifications can be gained by considering the object constants that it offers and the function constants and relation constants that are provided with each object type. SEMANOL supports several conventional object constant types, plus a few others that are appropriate to its specialized field of application. The Boolean, bit-string, integer, and string object types of SEMANOL are largely similar to those of other carefully designed languages, although strings are defined so that ordinarily non-represented characters, such as end-of-line tokens and line feeds, can be conveniently included. Sequences are ordered, possibly non-homogeneous, groups of items of any type; since these items may themselves be sequences, any type of hierarchical structure is easily modeled. Parse tree nodes in SEMANOL are equivalent to those of conventional representations, and syntax classes are likewise consistent with standard interpretations except for a context-sensitive means of denoting where optional spaces may appear.

Most of the SEMANOL operators are shown in Table 1, where they are grouped by the object constant type to which they apply. The operator names are generally meant to be self-descriptive, for ease of reading, and so should be roughly understandable without explanation. However, there are a few abbreviated operator names, as well as some unusual operators, that need some comment. The Boolean and bit-string operators are the conventional ones. SEMANOL integers are unusual in that they are actually numeric strings upon which string arithmetic is performed. Thus the range of SEMANOL integer arithmetic is unaffected by underlying machine factors and an idealized arithmetic is thereby realized. Note that division produces a truncated integer result, #NEG is a negation operator, and #CONVERT translates integers from one base (2, 8, or 10) to another. No other arithmetic is provided; our experience has been that floating point (and fixed point) semantics are best modeled through the use of integers. The concisely named #CW and #CS operators are string concatenation and sequence concatenation, respectively. The #SEG, #PARENT-NODE, and #ROOT-NODE operators are used to traverse parse trees, while #SEG-COUNT determines the number of immediately descendent nodes of a given node. The various #SEQUENCE collectors scan the parse tree in a uniform way so as to form a preordered sequence. The syntax class operator, %, is the Kleene star operator, with %1 denoting the case when at least one repetition is required.

A few SEMANOL operators are not easily classified and so do not appear in Table 1. #ASSIGN-LATEST-VALUE and #LATEST-VALUE are operators that store and recall information from what is, in effect, a single level associative memory wherein information is stored as (name, value) pairs. #GIVEN-PROGRAM reads a string, normally the defined language program to be interpreted, from one file, while #INPUT reads strings, commonly data input to the program being interpreted, from another file. #OUTPUT is a general purpose string output operator. The #CONTEXT-FREE-PARSE-TREE operator generates a parse tree representation from a given string and a given grammar. #STOP and #ERROR are used to terminate the interpretation process in various ways.

Type	Operators
Boolean	#AND, #OR, #NOT, #IFF, #IMPLIES
Bit-string	#BAND, #BOR, #BXOR
Integer	+, -, *, /, #NEG, #SIGN, #ABS, #CONVERT
String	#CW, #FIRST-CHARACTER-IN, #LAST-CHARACTER-IN, i #TH-CHARACTER-IN, #LEFT i #CHARACTERS-OF, #RIGHT i #CHARACTERS-OF, #SUBSTRING-OF-CHARACTERS i1 #TO i2 #OF, #PREFIX-OF-FIRST w #IN, #SUFFIX-OF-FIRST w #IN, #SUBSTRING-POSIT-OF w #IN, #LENGTH
Sequence	#CS, #FIRST-ELEMENT-IN, #LAST-ELEMENT-IN, i #TH-ELEMENT-IN, #INITIAL-SUBSEQ-OF-LENGTH i #OF, #TERMINAL-SUBSEQ-OF-LENGTH i #OF, #SUBSEQUENCE i1 #TO i2 #IN, #FIRST x #IN s #SUCH-THAT, #LAST x #IN s #SUCH-THAT, i #TH x #IN s #SUCH-THAT, #SUBSEQUENCE-OF-ELEMENTS x #IN s #SUCH-THAT, #REVERSE-SEQUENCE
Parse node	#SEG i #OF, #PARENT-NODE, #ROOT-NODE, #SEG-COUNT, #SEQUENCE-OF-NODES-IN, #SEQUENCE-OF-ANCESTORS-OF, #SEQUENCE-OF-NODES x #IN n #SUCH-THAT, #STRING-OF-TERMINALS
Syntax class	<...><...>, #U, #S-, %, %1

[Note: i = integer, w = string, n = node,
s = sequence, x = dummy]

Table 1: The SEMANOL Operators

The relational constants of SEMANOL are generally the expected ones, with natural extensions of equality and inclusion to parse tree nodes and sequences. An ordering relation, #PRECEDES, also applies to sequences and parse tree nodes. There are two quantifier relations included within SEMANOL,

#FOR-ALL...#IT-IS-TRUE-THAT... and

#THERE-EXISTS...#SUCH-THAT...,

that are very useful. The general inclusion of high level iterative facilities throughout SEMANOL permits expressive clarity to be provided the reader, and also aids the preparation of specifications.

The statement structure of SEMANOL is suggested in Table 2. The imperative commands are used for control and in procedural definitions. Most of a SEMANOL program is ordinarily given in a functional form that makes use of SEMANOL's recursive abilities; these functions are described by procedural or semantic definitions of the form shown. The rules for expression and condition composition in semantic definitions are typical of those of other programming languages. The syntactic definitions are used to define a grammar, and are only used in conjunction with the #CONTEXT-FREE-PARSE-TREE operator; the syntax expressions are similar to those of other grammar defining notations.

2.2 The Structure of a SEMANOL Specification

The foregoing suggests the information structures and operators provided by the SEMANOL metalanguage. However, a programming language definition expressed in SEMANOL is shaped not only by the facilities of the metalanguage itself, but also by the conventions and style that are adopted when writing the definition. Thus it is not enough to consider the metalanguage by itself; a better feel for SEMANOL can be imparted by explaining how it is used in describing a generalized specification.

In past projects, a standardized structure of specification was developed that should apply generally to many other common programming languages. This model is shown in Figure 1. The left side of the diagram shows the transformations which are made to the representation of the input program text as interpretation is performed. The central block diagram is a simple flow chart showing the series of processing steps that the SEMANOL

Commands

```
#FOR-ALL . . . #DO . . .

#WHILE . . . #DO . . .

#IF . . . #THEN . . .

#BEGIN . . . #END

#ASSIGN-VALUE . . .

#COMPUTE . . .

#RETURN-WITH-VALUE . . .
```

Procedural Definition

```
#PROC-DF name(param-1,param-2, . . . )

         command-1
            :
         command-n #.
```

Semantic Definitions

```
#DF name(p-1,p-2, . . . ) => expr #.

#DF name(p-1,p-2, . . . ) => exp-1 #IF condition-1 ;
                          => exp-2 #IF condition-2 ;
                          :     :      :         :
                          => exp-n #IF condition-n #.
```

Syntactic Definitions

```
#DF syntax-class => syntax-expr #.

#DF syntax-class => syntax-exp-1
                 => syntax-exp-2
                 :      :     :
                 => syntax-exp-n #.
```

Table 2: SEMANOL Statement Forms

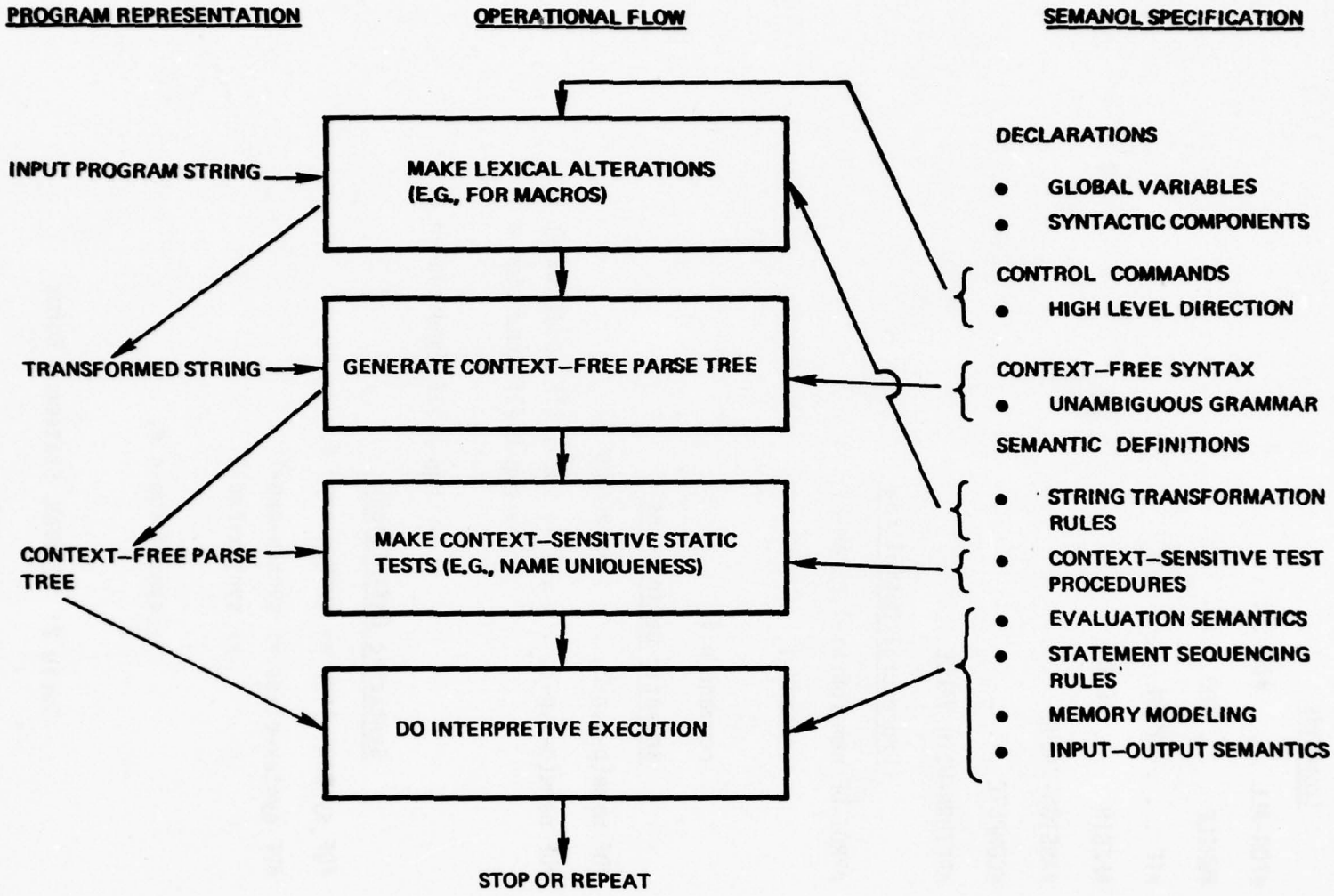

Figure 1: SEMANOL Specification Structure

metaprogram typically causes to occur. This flow is a consequence of having an operational specification method. The right side of the diagram reflects the static structure of a SEMANOL metaprogram and the way in which the various statement groups are related to the specification logic. This specification structure is considered in more detail in what follows.

The declaration section of a specification is composed of SEMANOL statements that identify global variables and syntactic components. Very few global variables are normally used in a SEMANOL specification; those of greatest importance are used to describe the control semantics of the defined language, and some are explained later. Syntactic components can also appear in this section; however, they are ignored in this report since they contribute to processing efficiency alone, and not to semantic description.

The control-commands section of a SEMANOL specification is conventionally composed of a few SEMANOL commands. These commands are executed sequentially and correspond to those available in conventional programming languages. Interpretation of the SEMANOL specification begins with the first statement of the control-commands section. These statements impose the overall control, as suggested in Figure 1; thus this section plays the role of a very high level main program.

The first step in describing a programming language is to define the lexical transformations that the language includes, if any. These transformations cause the source program to be altered by macro-like string substitutions (e.g., DEFINE in JOVIAL) or by textual deletions. While many programming languages require such a specification, other languages, such as BASIC, do not. Our current practice is to describe this feature of a programming language by use of the string operations of SEMANOL, with the operational consequence that the initial input source program string is altered to account for the occurrence of such statements in the input program. The transformed program text then becomes the basis for further specification. Incidentally, we have observed a tendency among language designers to specify these language features rather vaguely, seemingly from assuming an implied method of compiler implementation, but not describing it.

Following the textual alterations, the transformed program text is parsed. This parse process is invoked by the SEMANOL operator #CONTEXT-FREE-PARSE-TREE and is directed by the grammar given in the context-free syntax section of the specification. The product of this operation is a parse tree representation of the source program or one of two error conditions; the error conditions occur if the grammar can lead to more than one parse of the given program (i.e., the grammar is ambiguous) or if the program cannot be parsed. The parsed representation reveals the structure of the program and so is a convenient basis upon which to formulate the later semantic description. The lexically transformed program text itself is retained as the terminal leaves of the parse tree. The SEMANOL syntactic definitions used to define the grammar look much like the productions of the usual treatments of such programming language grammars. For instance,

```
#DF Program => <'START'> <#GAP> <Statement-list> <#GAP>
                <'TERM$'>#.

#DF Statement-list => <Statement> <%<<#GAP><Statement>>>#.
```

describes the syntax class Program as being initiated with a START, terminated with a TERM$, and containing at least one Statement. The #GAP set constant of SEMANOL is context sensitive; it is meant to describe the conventional rules that govern the use of blank characters in programming languages. The % states that zero or more occurences of <<#GAP> <Statement>> may follow.

Next comes the imposition of syntactic restrictions that cannot be expressed in a context-free grammar. That is, not all programs that can be parsed using the context-free grammar are legal programs in the defined language. It is the intent of this section of the specification to provide an operational algorithm to detect such illegal programs before an attempt is made to interpret them. The tests are commonly stated in terms of existence conditions, using the #THERE-EXISTS operator of SEMANOL applied

to the parse tree. The iterator and sequencing operators of SEMANOL, combined with the use of a parsed representation, allow these restrictions to be expressed succinctly. Since the application of these tests is made before interpretation proper, they can cause the rejection of programs that could be interpreted without encountering the error condition. That is, these tests correspond to those that a compiler might make and the consequences can be semantically different than if similar tests were applied at execution time (e.g., consider unexecuted references to undefined variables). Typical tests included here are those for name uniqueness, type conformity, formal and actual argument agreement, program accord with structural rules, data organization consistency, etc. While SEMANOL enforces these limitations readily, the formulation of rules for syntactic exclusion is a very difficult problem in language design and one that is ordinarily given limited attention.

Having specified these preliminary operations, the semantics of program execution are next given. Figure 2 illustrates the data structures that are used to explain execution semantics. The parse tree is (effectively) a static structure that is traversed and tested by direction of the SEMANOL specification; the SEMANOL specification then causes the execution effect of the parsed program to be realized. The semantics of the SEMANOL specification of a defined language are thus generally expressed in terms of the grammar that is used to create this parse tree. Because of this, there is a close relationship between the context-free syntax section and the semantic definitions section of a SEMANOL specification. The storage mechanism of SEMANOL is used to record the changes that occur as the interpretation is performed. Note that this overall model corresponds rather closely to that of any conventional computer; this similarity emphasizes again that SEMANOL provides an operational description of the programming language being defined.

While there are several interwoven facets to this description, it seems best to first consider how the semantics of storage modeling are expressed. The storage mechanism of SEMANOL provides a means for giving a concise explanation of storage semantics for defined languages that view data in an abstract manner, as does Minimal BASIC. Unfortunately,

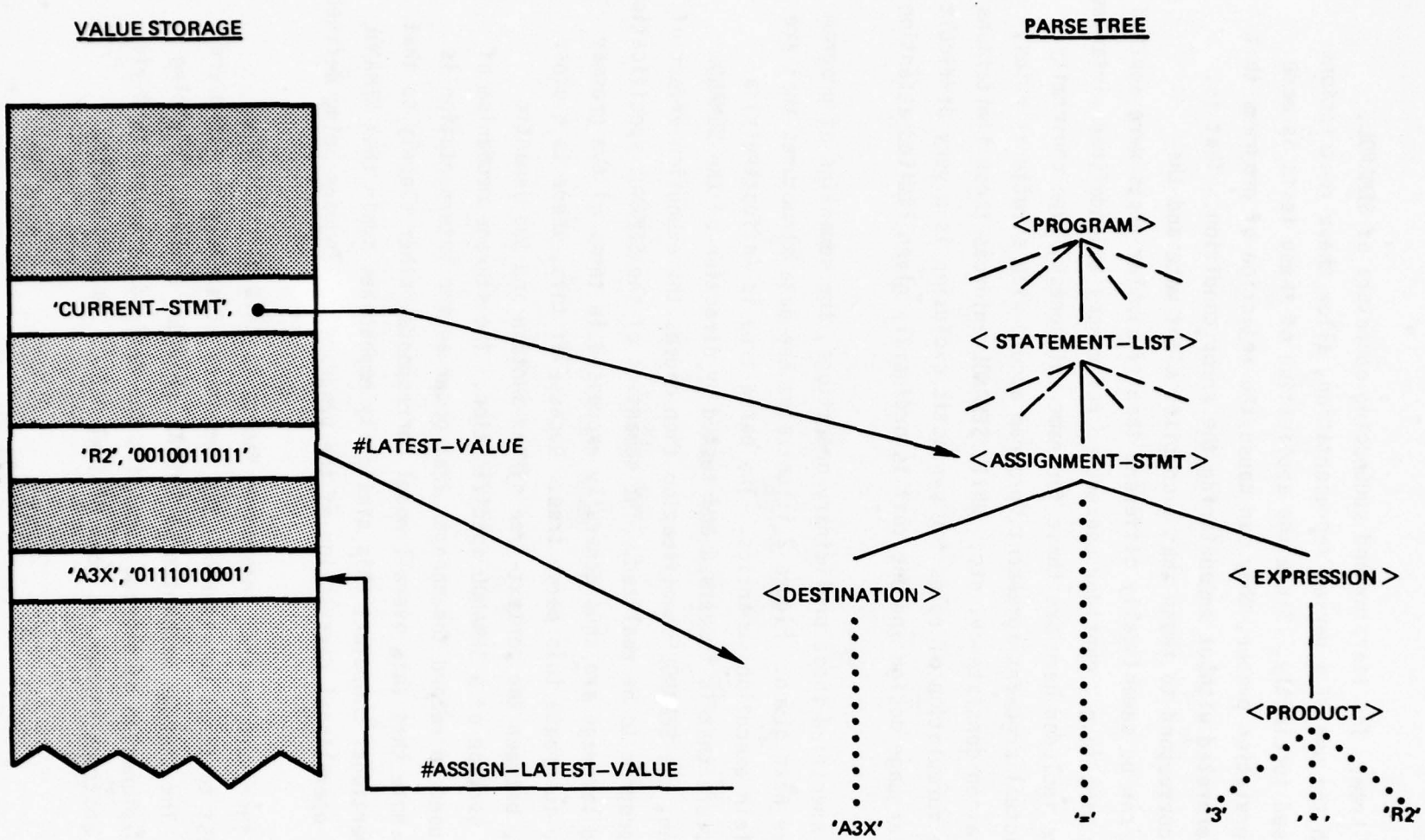

Figure 2: The Data Structures of Interpretation

many present languages do not adopt an abstract viewpoint. In particular, languages, like JOVIAL, that permit unrestrained use of pointers and data overlay demand that storage be described in the hard realities of bits and computer words. The effect of this is that the names used for storage and retrieval become (equivalent to) storage addresses and that the values associated with these names become the binary contents of the addressed locations. A form of storage allocation is thus defined so that these storage addresses can be established. The SEMANOL specification thereby becomes appropriate only for a given language implementation; however, this condition is inescapable if the semantics of such languages are to be completely defined. And, of course, completeness is demanded if the specification is to be processed by the Interpreter. But note that a family of implementations can sometimes be created through parameterization of the storage model semantics.

Closely related to the storage model is the semantic definition section that explains the evaluation rules for the language being defined. Here are explained the semantics of arithmetic operations of the types (e.g., integer, floating, fixed, location) provided in the defined language, character string operations, Boolean evaluation, and comparison relations. The description of evaluation may include consideration of machine effects upon the computational result produced, type conversions that may be required, and a suitable interface with the storage model. The general evaluation procedure followed is largely one of translating the constants and operators of the defined language to corresponding SEMANOL representations, and then performing the computation upon the SEMANOL constants. Evaluation is conveniently described recursively and SEMANOL's recursive abilities are used advantageously here.

The top level control flow semantics for the defined language are commonly given in a SEMANOL semantic procedure that is first invoked when interpretation of a defined language program is begun, and subsequently

called recursively whenever procedure and function calls are to be interpreted. The rules of statement sequencing, procedure and function invocation and return, loop control, and branching, are then given by subsidiary semantic definitions. The control semantics of the defined language are, by convention, stated in terms of the declared SEMANOL global variable CURRENT-STMT. CURRENT-STMT holds the node of the executable unit (e.g., statement phrase) currently being processed. The active point in the defined language program is defined by this value, transitions of control by changes in this value. Note that the SEMANOL metalanguage itself does not include any form of defined language control model; the way in which control is to be defined is totally the responsibility of the SEMANOL programmer. Also note that the control semantics of procedure and function calls can become very much intertwined with evaluation if the language being defined permits short-circuit evaluation, abnormal returns, and other features that disrupt orderly expression evaluation.

The semantics of input-output deal with data transmission and with translations between internal and external representations of data. Any translations required are accomplished by the string operators of SEMANOL. The transmissions are accomplished by the #INPUT operator and the #OUTPUT operator. In this way, input-output semantics relating to the external world can be modeled. Input-output that is internal to a program in the defined language can be described through the use of sequences and a suitable set of representation conventions.

2.3 The SEMANOL Interpreter

The SEMANOL Interpreter accepts a SEMANOL specification of a programming language and uses that input specification to realize the semantic effect of (i.e., to execute) programs written in the language thus defined. By virtue of the Interpreter, SEMANOL specifications can themselves be tested and debugged. Furthermore, an operational standard for the defined language is thus created.

The operation of the elements that constitute the SEMANOL system is shown in Figure 3. The broken line encloses the SEMANOL Interpreter, which can be seen to actually consist of two loosely connected programs identified as the Translator and the Executer. The Translator processes the SEMANOL description of a programming language and so generates an intermediate code form known as the SEMANOL Internal Language (SIL). This translation phase also tests the SEMANOL description for its syntactic correctness, much as a conventional compiler would do, and continues only for acceptable descriptions. The SIL representation corresponds to a list of operators and operands, and direct transfers, that are used to control the Executer. The Executer program is essentially a stack oriented processor that then interprets the SIL code and so performs the operational interpretation of a program in the defined language. As shown in Figure 3, this interpretive processing commonly includes the reading of input and the production of output at the direction of the defined language program being processed. Note that the SIL code is presently recorded in a character format in order to (1) minimize the interface requirements upon the Translator and Executer and (2) permit the SIL file to be read and manipulated by the standard editing facilities of the host operating system.

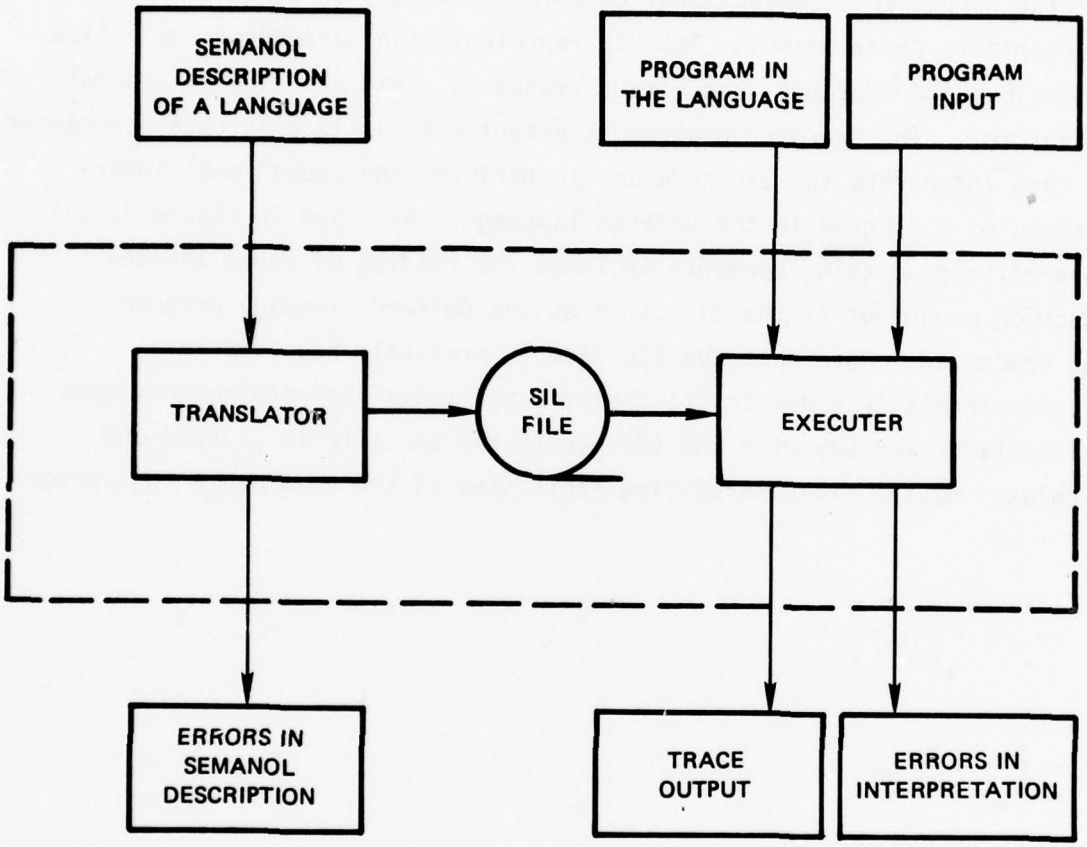

Figure 3: SEMANOL Interpreter Organization

3. THE GENERAL APPROACH

This section describes the basis on which compiler test cases are to be generated by the system designed in this project, and gives a brief introduction to the system design itself. A more detailed explanation of the system design will be found in Section 4 of this report.

3.1 A Basis For Testing

The goal of this project was to develop an automated method of generating programs that would form an effective test of a compiler for a given language. It was based upon using a formal SEMANOL specification of the given language as the basis for the automatic generation, and upon use of the SEMANOL Interpreter programming system as a means of measuring the effectiveness of the test cases generated. This structure is shown in Figure 4. Since the formal SEMANOL notation used for semantic description, unlike the prose text commonly used in programming language specification, can be "understood" by a computer (to the same extent as can any other programming language), it provided a needed prerequisite for automatic test case generation.

The test cases to be generated by the methods discussed in this report are thus to be derived from the SEMANOL specification of a programming language. Their effectiveness will also be measured relative to that specification. That is, the test cases to be generated are intended to be a satisfactory test of the metaprogram that is a SEMANOL specification of a programming language. The presumption is then being made that this set of test programs (for an interpretive language processor) will be rather similarly effective when used with a compiler implementation for that same language. In some sense, this test procedure is based on showing that a compiler implementation is equivalent to an interpreter implementation, the SEMANOL metaprogram, that is taken to be correct. The presumption that the SEMANOL specification is correct is a convenient bias to adopt, even though

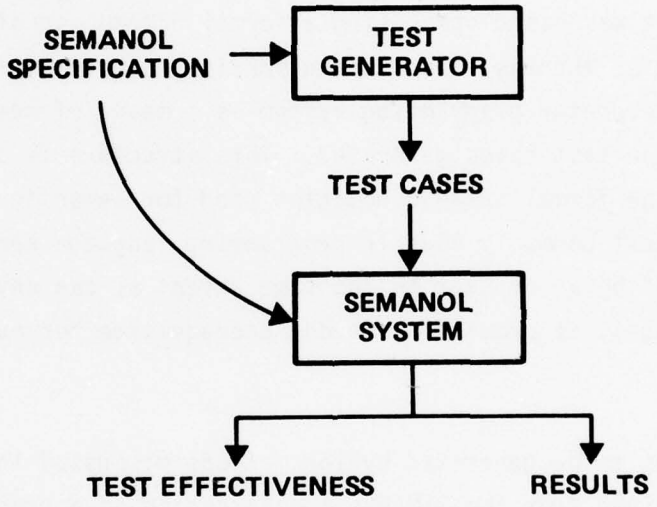

Figure 4: The Test Generation System

its "correctness" can only be guessed at through testing, and so will be slightly suspect in practice. Nevertheless, because a SEMANOL specification is a formal one, compiler testing is done against a precise standard and not against the usual vague requirements that are commonly encountered in acceptance testing.

Test effectiveness in this approach is measured in structural terms with regard to the SEMANOL metaprogram; e.g., in terms of SEMANOL definitions exercised, execution paths traced through the SEMANOL specification, etc. It is then argued that a structural test of the SEMANOL metaprogram is a similarly effective test of a compiler. The argument is simply that:

1. Both the compiler and metaprogram are meant to perform the same function.

2. A good test of one function should be a good test of the alternate implementation.

3. Thus a good test of the metaprogram will be a good compiler test.

The weakness in the argument rests with the uncertainty as to what a "good" test is. It is recognized that the structural coverage (e.g., proportion of statements executed at least once) for one implementation will not ordinarily be the same as for another, yet it would seem reasonable to expect that high coverage for one would be reflected in a similarly high coverage for the other. It is on the basis of such programming common sense that we base our notion of test effectiveness. (But do note that other measures of test effectiveness, perhaps in structural terms related to the compiler, should be determined, where possible, for the test set so that test set quality can be measured in other ways. Our notion of test effectiveness is meant to guide the generation system, not to provide an exclusive measure of testing capability.)

Observe that this approach is intended to create tests of features of the language, but not of important compiler characteristics such as those related to generated code efficiency or size, compiler error recovery, compiler limits, etc. There are thus important compiler attributes for which tests will not be created explicitly, since these features are not ordinarily described in the SEMANOL specification; however, assistance in generating such tests can be provided by the automated system described herein. Also note that compiler validity is determined by observing the effects of executing compiled programs, and not by inspection of the compiler generated code itself to determine if it is a faithful translation of the given source language test. To look at the object code is a machine and implementation dependent process, while the evaluation of execution effects deals with semantic consequences with a much less direct involvement in implementation dependencies. This interest in results, not generated code, is the conventional view, and one adopted by efforts to create standard test sets for any programming language (e.g., COBOL and FORTRAN).

Measures of test effectiveness will be gathered through use of the SEMANOL Interpreter, suitably modified along the lines described later, and so are based on structural coverage of the metaprogram specification. We have considered various test objectives, such as executing each definition at least once, exercising every branch possibility, exercising every metaprogram execution path, etc., and have settled upon total definition coverage as being a useful, but relatively simple, test goal that is consistent with our preliminary stage of development. The effect of choosing a test goal is felt in the Analyzer Part of the test generation system, since it is there that test selection criteria may influence the process.

3.2 The TRW Approach To Test Generation

It can be seen that this overall process is one in test case derivation, which is distinctive in that the test data sought here are themselves programs. The overall approach to automatic program generation developed in this project, suggested in Figure 5, is to divide the process into Analyzer and Synthesizer phases. This division is convenient from a

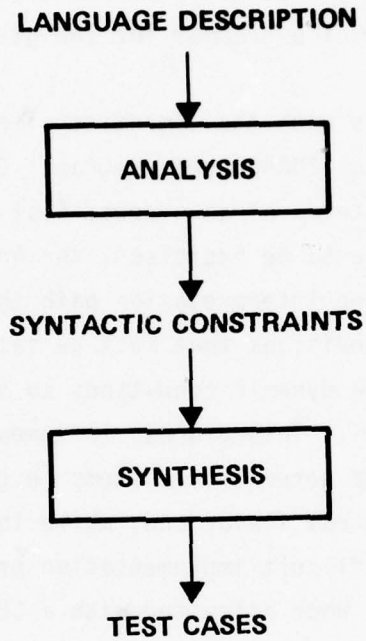

Figure 5: A Test Generator Design

design viewpoint in that the total problem is thereby broken into conceptually more manageable parts; it also has the special advantage that the Synthesizer can be used independently in the test generation process. Thus this choice extends the overall usefulness of the proposed system. The role of the Analyzer phase is to transform SEMANOL specified run-time semantics into syntactic characteristics of a program such that a given execution effect would be realized. These syntactic properties are expressed as constraints upon program generation from a context-free grammar for the given language. It is the Synthesizer phase that performs this constrained generation of a program from the grammar for the given language.

The Analyzer thus deals with the semantics of execution as these semantics are expressed in a SEMANOL metaprogram. Given a semantic effect to be tested, expressed in terms of an ordered list of elements of the SEMANOL metaprogram that are to be exercised, the Analyzer is meant to determine a suitable covering interpretation path through the SEMANOL metaprogram, collect the conditions that must be satisfied for this path to be followed, and solve these dynamic conditions so as to produce essentially static syntactic constraints. This process is somewhat like that being attempted in other test data generation systems (e.g., [Clarke 1976], [Boyer et al 1975]) in general flavor and, while the process sounds simple enough, it presents very difficult implementation problems even in simplified circumstances and certainly when attempted with a SEMANOL metaprogram defining a contemporary programming language. In particular, the test data to be found here constitutes the very complex structure that is a computer program, and not the simple values that other systems seek.

The Synthesizer phase is essentially one of using the context-free grammar for the given language in a generative manner to produce programs, but with the generation being governed by the Analyzer-produced constraints (including the context-sensitive restrictions of the language itself). This constrained generation will then produce program instances that test specific features of the language being tested. The constraints ordinarily

are meant to be satisfied by many possible programs; the Synthesizer makes the selection concrete. This process is conceptually rather simple, more so than the Analyzer, and the major difficulty in realizing an implementation is in overcoming computational problems.

For neither the Analyzer process of path determination nor the Synthesizer process of program generation do analytic solutions exist; thus direct construction of output results cannot be done. As with other processes in AI, it then becomes necessary to somehow generate potential solutions and to test each to see if indeed it meets the conditions required of an acceptable result. This type of process demands that a method for systematic generation of solution possibilities be developed to define the search space (and requires that a scheme of search space representation be correspondingly developed). For the Analyzer process, the solution space consists of a set of some form of metaprogram execution path representations; for the Synthesizer, the solution space is made up of a set of derivation (i.e., parse) tree representations. Since the SEMANOL metaprogramming language includes control loops and recursion and the context-free grammar is also recursive, the number of possible solutions is infinite in both cases and the length of an individual solution candidate is also unbounded. For that reason, search procedures obviously cannot consider all possibilities; the emphasis thus is placed upon finding ways in which solutions can be found quickly. Algorithms must include search stop rules, if algorithm termination is to be assured, and this condition places great importance upon the way in which a given problem is formulated; one form of problem statement may be satisfied by a given search algorithm while an alternate formulation may not be satisfied within the search limits. It is also obviously critical that algorithms must be able to avoid considering impossible (or even unlikely) cases and that retreat from eventually fruitless possibilities be intelligently done. Such considerations have received much of our effort in this project.

This overall process is expected to be one of iteration. The Analyzer creates constraints, from which the Synthesizer produces a test program, whose test effectiveness is measured by the Interpreter program. The effectiveness measure of this test is combined with that of previously generated test programs and used to decide what form of test should be constructed next. The form of the next test to be generated is represented in Analyzer input, and the process repeated until a test set of adequate effectiveness has been collected.

The Synthesizer, as already noted, is a program that generates compiler test programs from a context-free grammar for the defined language, with this generative process being subject to constraints that are meant to guide the process to the creation of useful compiler test programs. To be practical, it is necessary that the Synthesizer program create its test program within a reasonable computational time. Note that the test cases generated are the result of a program derivation process (as in top-down parsing), and that a context-free derivation tree is created, by the Synthesizer, whose leaves are then collected to form the desired program text. The constraints are implicitly expressed upon the root node of the derivation tree and hence upon the start symbol for the context-free grammar.

A most obvious approach to Synthesizer design is to consider a program that would be able to systematically generate programs of the defined language and, as a program was generated, would test each for constraint satisfaction. Since context-free grammars commonly define an infinitude of programs, and certainly more than one would want to have to consider, such a process would undoubtedly include some form of computational stop rule; failure to find a solution within that time would then be cause for making some revision in the constraints before another effort was made. One has essentially a "create and test" system with this approach. The major problem with this method is that incorrect choices in the derivation process are not recognized until complete trees have been created.

Thus a bad decision choice made early in the development of a derivation tree is only realized after a great deal of fruitless computation has been performed. The algorithm designed in performance of this contract makes use of a scheme of partial evaluation and constraint propagation so as to decide the acceptability of a derivation before a complete tree is necessarily generated; a computational improvement is thus provided.

In brief, the idea of partial evaluation and constraint propagation is that the initial root node constraints can be transformed (i.e., factored) into derived constraints applicable to nodes of a partial derivation tree being constructed. The derived constraints are formed by evaluating constraints on the partial tree as each expansion step is performed. If such propagated constraints,as determined by partial evaluation, cannot be met at a given node (i.e., a constraint evaluates to false), it is immediately known that no further derivation efforts along that subtree need be attempted. Thus incorrect derivation decisions can be recognized more quickly. If the constraint is met (i.e., evaluates to true), further propagation is not needed; a solution has been reached. If the constraint cannot be evaluated, because the partial tree is not yet adequately expanded, the given constraint is modified to become a set of constraints that apply to unexpanded nodes in the tree so far generated. Note that the nature of propagation means that satisfaction of the derived constraints guarantees that the original root node constraints are thereby satisfied. Note that constraints, as they are passed through the developing derivation tree, are partitioned so that not all constraints need be tested at each node; further computational economy is thereby achieved.

The synthesis process builds a succession of partial derivation trees until it creates one whose leaves are all terminal symbols; these leaves then form a (context-free)syntactically valid program in the language for which tests are being sought. In achieving this, successive partial trees are each formed from their predecessor by applying a production of the grammar to a non-terminal node to create an expanded tree.

Each expansion step ordinarily involves a choice among production cases and any one, because of the need to satisfy constraints, may be an incorrect choice. (We have invested considerable effort in trying to find ways in which this selection process can be made successful.) Whenever it is realized that the derivation tree so far formed cannot satisfy the constraints, the tree must be modified and derivation resumed from this restored basis; this process is referred to as backtracking, or backup, since one goes back to a previous state before continuing the search for a solution. As noted already, bad decisions will often not be detected until many other intervening decisions have been made; the intention is to jump back to the bad decision directly rather than working backwards through all the intermediate points, exhausting the possibilities of each to no useful effect, etc. The heuristics which appear hopeful for this backup process are given later.

Constraints are used to cause the Synthesizer to generate a specific type of program (or even a specific program). Ordinarily, they state general characteristics wanted in a test program to be generated, and the Synthesizer is then left to provide a specific instance of such a program. In the implementation contemplated here, constraints are boolean-valued expressions of at most one free variable, that are associated with the total program to be generated or some part of it. We have found the SEMANOL notation to be well suited to stating constraints. Constraints are then augmented by constraint definitions; these definitions are also expressed in SEMANOL notation, as semantic functions (#DF's) or procedures (#PROC-DF's), and so can compute subsidiary values needed when evaluating constraints. As these constraint definitions apply to partial derivation trees, rather than complete parse trees, certain restrictions on their structure are necessarily introduced and an ability to denote that their evaluation cannot be completed has been added. However, most of SEMANOL's power can be used in expressing constraints. Because of this, constraints are able to state conditions that, while still syntactic in character, are far more than simple restrictive grammatical rules imposed on the context-free syntax.

The Analyzer creates constraints in an overall process of test generation that is one of iteration. Whether the process is begun with a collection of rather random "seed" programs or by some initial programs generated through an established algorithm, and regardless of the SEMANOL based measure of test effectiveness being applied, test programs eventually will be wanted whose interpretation (literally by the SEMANOL Interpreter) will cause certain parts of the SEMANOL metaprogram to be activated (i.e., exercised).

The parts of the SEMANOL metaprogram that can be executed are called structural elements, and each is represented by a node in the execution path. The input to the Analyzer is thus stated to be a sequence of nodes that a test program is to exercise in a given order. The role of the Analyzer is to find a legitimate execution path through the SEMANOL metaprogram that covers the designated structural element nodes in the required order, and to translate the conditions that will cause this path to be followed into syntactic constraints. If done properly, the program generated by the Synthesizer in response to these constraints will produce an Interpreter trace that agrees with the path the Analyzer constructed. In effect, the Analyzer is simulating SEMANOL metaprogram execution to find a path that includes certain execution events.

The path structure used by the Analyzer is composed of primitive event and decision nodes, plus nodes that represent metaprogram function invocations. In this structure, loops are expanded and the loop appears in linear form with the prerequisite number of node-group repetitions. Each function is then given a flow graph representation. The generation process is begun by creating some path through a selected metaprogram function, with the path being one that is determined by (i.e., derivable from) the flow graph for that selected function. A node that represents a function invocation in this path is then expanded, or replaced, by a path derived from the flow graph for the function invoked. The choices

at each expansion step, of the path node to expand and of the particular flow graph derived path to be used in the expansion, are made so that they contribute to reaching the nodes that are to be covered. As nodes are expanded, the node coverage requirements are revised to account for the subpaths thereby created; the efficiency of the path building process is improved as a consequence. This generation part of the Analyzer process can be seen to be similar to the way in which the Synthesizer generates a derivation tree; invocation nodes correspond to the non-terminal symbols of the grammar, while flow graphs correspond to the production of the grammar.

However, an added condition is placed upon an execution path that has no counterpart relative to Synthesizer derivation trees; namely, that the execution path be semantically legitimate. That is, the path the Analyzer constructs must be one that can be followed by a valid program when it is processed by the SEMANOL Interpreter. This means that the execution conditions associated with the path must be consistent (i.e., not mutually contradictory). The Analyzer algorithm excludes inconsistent paths through the use of a form of symbolic execution. In symbolic execution, input variables are retained throughout execution as dummy symbols, rather than being given actual values as they would be ordinarily (e.g., a computation might be done with 'a' rather than the expected integer input value, perhaps 2). Symbolic computations are then performed to yield symbolic expressions, and to record the conditions upon the input to achieve the resulting output. Symbolic execution leads to a symbolic state that is incrementally updated as each path extension is made. The new symbolic state is then tested to determine its consistency; if the expansion of the new subpath has created a property that cannot hold simultaneously with some other property of the symbolic state, the subpath is inconsistent with the path constructed so far. Subpaths leading to inconsistencies are then rejected and alternatives tried; that is, backup is induced.

As a consequence of using symbolic execution in the derivation of an execution path, the conditions that must be satisfied for this path to be followed are already part of the symbolic state. The path conditions in the symbolic state are given as SEMANOL boolean expressions that apply to symbolic designators standing for nodes in the parse tree of the test program whose execution path has just been found. The boolean conditions themselves generally refer to syntactic properties of this implied test program. Given these properties of path conditions, it is relatively easy to transform them to syntactic constraints that apply to the root node of the parse tree; in this form they are passed on to the Synthesizer (along with the subsidiary syntactic definitions that are needed to complete the definition of the constraints).

We should note that determining path consistency is a problem that, in general, does not have an effectively computable solution. However, the SEMANOL metaprogram that is being treated here is not an arbitrary program; it is highly structured, and that structure can be varied somewhat if it is found helpful to do so. Because of that, it is believed that adequate consistency checking heuristics can be developed, although further research will be needed to confirm this belief.

3.3 Observations On This Approach

It should be observed that the procedure described will produce test cases that are known to:

1. Conform to the context-free grammar of the language being tested. The Synthesizer uses this grammar in its node expansion process in such a way that it can only generate legal derivation trees; thus all programs produced will conform to the context-free grammar.

2. Satisfy the context-sensitive restrictions of the language. Whether these restrictions are a part of the general set of Synthesizer constraints or derived specifically for each case by the Analyzer makes no difference; these syntactic restrictions of the language under test will be met by any test case produced.

3. Run to completion without error when executed. The Analyzer only constructs paths for test programs that will terminate correctly (i.e., without #ERROR).

While this generally means test programs will be correct programs, it does not rule out the automatic generation of test programs that deal with some run-time error conditions. Machine and environmental factors do appear in SEMANOL specifications, so it is possible for the Analyzer to find constraints for a program that induces a run-time error condition, such as arithmetic overflow, provided the SEMANOL specification also defines the semantic response (such a program is not formally incorrect). However, many other run-time errors, and most compile-time errors, would ordinarily be ignored in a SEMANOL specification. For these, our fully automated procedure would not provide tests; however, the Synthesizer can be used by itself as a generator for tests whose constraints are formulated by the user. Generally speaking, it is felt that this method of developing tests of error treatment is to be preferred over attempting to design an entirely automated procedure.

There are also potential problems in testing that are related to the optimization of generated code by compilers. For instance, the constant expression 2+3 might be evaluated during execution by a load, add instruction sequence generated by a compiler. However, an optimizing compiler would very likely recognize that a constant value was being computed, and so would not generate code to perform the addition but, instead, would itself compute the sum 5 and make that available during execution.

The difficulty in this is that both compilers will presumably yield the same answer to a test case calling for such a sum, but will have done so in a much different way. Both methods are certainly right in input-output relational terms. However, the inferences that can be drawn from successful execution of this test case are not the same in both cases. While it may be reasonably justifiable to assume that correct code will be created for summation generally (e.g., for variable operands) in the non-optimizing case, such an assumption cannot be made in the optimizing case since no generated code was tested. Different aspects of the two compilers have been tested by a single test case.

In the face of compiler optimization, it is often difficult to determine whether a test has been made of the code generation facilities of a compiler or its optimization facilities. Generally, tests are meant to test code generation accuracy; it is that which our tests are likewise meant to (primarily) examine. This means that the test cases generated must be structured so that they are unlikely to be optimized out of existence. Since we would prefer test cases to be rather simple and to generally operate upon constants (i.e., to be potentially computable at compile-time) for ease of making them self-checking, there is some problem. The solution we plan is to impose additional constraints upon the test cases that are generated, with such constraints being determined for the specific compiler to be tested. These constraints would be similar to the context-sensitive restrictions of the defined language and form an adjunct to them. Their propriety would be determined through actual examination of the object code produced by the compiler under test. Note that the ease of imposing such conditions upon test cases generated by the Synthesizer is another illustration of the power of an automated system of test case generation.

4. A TEST GENERATOR DESIGN

It is very important to realize that the compiler test case generator system design presented in this section is meant to provide a framework in which implementation and experimentation can be reasonably pursued. The design is given in general terms and often discusses alternative implementation options. The emphasis is on proposing an automated compiler test generation procedure that can be implemented with ease and is readily changed, that is understandable, and that is adaptable; the proposed system is meant to encourage experimentation in order that better formulations of the test case problem can be found, improved heuristics developed, ever greater automation achieved, etc. It is imagined that a number of improvement phases will be needed to produce a production system; only a first step in this process has been taken in performance of this contract.

This desire to be practical has sometimes led to suggestions that user participation be provided in the design even though heuristics to accomplish similar things can be imagined. In general, this has occurred where the value of the known heuristic is uncertain and the cost of implementation is obviously high.

4.1 The Synthesizer

The Synthesizer is a program that generates test case programs, from a given grammar, that meet given syntactic requirements, or constraints. The design presented here is one that was developed to lend itself to an initial implementation at reasonable cost, when the simpler heuristics are used, and that can be incrementally extended, as discussed in this section, as the need for such extension is established. The design provides a useful implementation framework for experimentation and further research.

The strategies and heuristics proposed for the Synthesizer are largely adapted from those that have appeared in the AI literature. However, this material ordinarily did not give general principles and certainly was not

The overall nature of the CBS algorithm is one of expanding a (syntax class) node of the partial tree by choosing a production case for that syntax class, thereby extending the partial tree one step; then choosing the next node to expand from those unexpanded nodes still in the partial tree, including any unexpanded nodes just created by the immediately prior expansion step; then expanding the node thus selected; and on and on. A tree is thus built by expanding nodes in accordance with the synthesis grammar and constraints given.

The top level program of this algorithm, called CBS-SYNTHESIZE, is given in Program Design Language form in Figure 6. The program first reads the synthesis grammar, the set of constraints, and the set of constraint definitions. A partial tree is formed that consists of a single node, the root node of the synthesis grammar; the given input constraints are then bound to this root node. CBS-EXPAND-NEXT-NODE is then called to set the solution search in motion; it will return only after a solution tree has been found, the tree returned then being complete, or the search failed to find a solution, in which case the tree returned is incomplete.

The CBS-EXPAND-NEXT-NODE procedure, Figure 7, is used recursively, with an invocation of CBS-EXPAND-NEXT-NODE being created for each node that is being expanded. The procedure achieves exhaustiveness by cycling through all possible case expansions (selected through POSSIBLE-CASE-LIST and generated by the COMPUTE-POSSIBLE-CASE-LIST function) for its argument node before backing up. At each expansion step, the augmented tree is tested to determine its compliance with the constraints that apply to the node being expanded. Constraints are distributed through the partial tree by PROPAGATE-CONSTRAINTS-FROM, while COMPUTE-PARTIAL-EVAL-RESULTS determines constraint truth or falsity at the argument node and also reduces the constraint expression for the case selection made.

```
*****************************************************************************************
*                                                                                       *
*  1    READ SYNTHESIS GRAMMAR                                                          *
*  2    READ ROOT NODE CONSTRAINTS                                                      *
*  3    READ CONSTRAINT DEFINITIONS                                                     *
*  4    CONSTRUCT INITIAL-PARTIAL-TREE WITH ROOT NODE                                   *
*  5    ADD ROOT NODE CONSTRAINTS TO INITIAL-PARTIAL-TREE                               *
*  6    RESULT = CBS-EXPAND-NEXT-NODE (INITIAL-PARTIAL-TREE)                            *
*  7    IF NOT IS-COMPLETE(RESULT)                                                      *
*  8        OUTPUT 'FAIL'                                                               *
*  9    ELSE                                                                            *
* 10        OUTPUT CONVERT-TO-OUTPUT-FORM(RESULT)                                       *
* 11    ENDIF                                                                           *
*                                                                                       *
*****************************************************************************************
```

Figure 6: CBS-SYNTHESIZE Procedure

dealing with this specific problem. The basic algorithm used here is an improved version of the general backtrack paradigm presented in [Golomb and Baumert 1965] and [Bitner and Reingold 1975]. While the idea of constraint propagation was prompted by [Stallman and Sussman 1976], the use of partial evaluation, although it has appeared in other contexts (e.g., [Lombardi and Raphael 1964]), in constraint propagation is thought to be new. The approach appearing here for expansion order and case selection is a composite product drawn from many sources; however, the ideas of least commitment in [Sacerdoti 1977] and least constraint in [Mazlack 1976] were especially influential. Backup methods discussed here drew heavily upon the work of [Sussman and Mc Dermott 1972] and [Stallman and Sussman 1976] in attempting to design backup so as learn from errors and to make use of dependency relationships.

The use of chronological backtracking was adopted in the Synthesizer because its clarity provides an implementation and conceptual model that is readily understandable. We believe that a workable system can be developed in this framework. (Note that [Gerhart and Yelowitz 1976] discuss ideas that apply to proving assertions about backtrack algorithms.) It is recognized that this type of backtracking is imperfect (e.g., [Sussman and Mc Dermott 1972]) in that it undoes decisions that could be retained. However, we do suggest ways in which the effect of prior decisions can be retained for use in tree regeneration after backup, and an inherent problem in chronological backtracking is thereby partially overcome. It is our opinion that a Synthesizer using the backtracking methods we propose can achieve adequately efficient performance.

The following discussion uses several terms that need definition. A derivation tree contains nodes that represent either terminal or non-terminal symbols of the synthesis grammar; nodes associated with terminal symbols will be called <u>terminal nodes</u>, while nodes associated with non-terminal symbols will be called <u>non-terminal nodes</u>. A <u>complete (derivation) tree</u> can then be defined as a tree whose leaf nodes are all terminal nodes, while a <u>partial tree</u> is one that contains one or more leaf nodes that are non-terminal nodes. A non-terminal leaf node is referred to as an <u>unexpanded node</u>, since it must be expanded before a complete tree can be obtained.

This means that a complete derivation tree is one that does not contain unexpanded nodes. An expanded node is then a non-terminal node that is not a leaf node; that is, an expanded node is an interior node (by virtue of the fact that a case selection has already been made for it).

The Algorithm

A simple Synthesizer algorithm would generate complete derivation trees in some fixed order, testing each until one satisfying all constraints was produced. Unfortunately, this scheme is computationally unfeasible. Our Constrained Backtrack Synthesis (CBS) algorithm attempts to be computationally efficient enough to be practical. It seeks to do this by evaluating constraints on the derivation tree as it is being generated (and so is only a partial tree) rather than testing for constraint satisfaction only after a complete tree had been generated. By evaluating constraints as each new partial tree element is generated, it becomes possible to quickly detect decisions that cannot lead to satisfactory derivation trees; thus large parts of the potential search space can be eliminated from consideration and computational efficiency thereby enhanced.

The algorithm also attempts to improve performance through a method of constraint propagation (and its related system of partial evaluation). When dealing with root node constraints and complete trees, it is natural to apply the constraints to the complete candidate tree and so to decide whether this candidate tree was a solution or not. When dealing with partial trees, of course, it also is possible that a constraint will depend upon an unexpended node for its value and so have an undetermined result. But in any case, as long as constraints are applied strictly to the root node, all the constraints must be evaluated for each new case of node generation, an unappealingly slow procedure. Thus constraint propagation is introduced to distribute the constraints through the partial tree in order that only those constraints pertinent to a particular part of the subtree need be evaluated at a given node. The nature of the propagation method is such that if the propagated constraints are satisfied, then the original root node constraints are also certain to be satisfied.

```
IF IS-COMPLETE(PARTIAL-TREE)
    RETURN PARTIAL-TREE
ENDIF
CURRENT-NODE = NEXT-NODE-TO-EXPAND-OF(PARTIAL-TREE)
COMPUTE-POSSIBLE-CASE-LIST(CURRENT-NODE)
DO FOR CURRENT-CASE = EACH ELEMENT OF POSSIBLE-CASE-LIST(CURRENT-NODE)
    EXPAND(CURRENT-NODE,CURRENT-CASE)
    IF NOT EXCEEDS-LIMITS(PARTIAL-TREE)
        COMPUTE-PARTIAL-EVAL-RESULTS(CURRENT-NODE)
        IF NO-PARTIAL-EVAL-RESULT-IS-FALSE(CURRENT-NODE)
            PROPAGATE-CONSTRAINTS-FROM(CURRENT-NODE)
            RESULT = CBS-EXPAND-NEXT-NODE(PARTIAL-TREE)
            IF IS-COMPLETE(RESULT)
                RETURN RESULT
            ENDIF
            UNPROPAGATE-CONSTRAINTS-FROM(CURRENT-NODE)
            IF RESULT NOT= CURRENT-NODE
                CONTRACT(CURRENT-NODE)
                RETURN RESULT
            ENDIF
        ENDIF
    ENDIF
    CONTRACT(CURRENT-NODE)
ENDDO
RETURN SCAPEGOAT-NODE(CURRENT-NODE)
```

Figure 7: CBS-EXPAND-NEXT-NODE Procedure

Detection that an expansion step will lead to a failure, as shown by an element in the PARTIAL-EVAL-RESULT-LIST being #FALSE or the EXCEEDS-LIMITS condition being met, causes backup. The selection of the decision point from which to restart the search is made by the SCAPEGOAT-NODE function, which returns the node to which retreat is to be made. The recursive chain of CBS-EXPAND-NEXT-NODE invocations is then unwound until the invocation is found that was expanding this node (i.e., the invocation for which this is the current node). CONTRACT and UNPROPAGATE-CONSTRAINTS-FROM do the step-by-step reduction of the partial tree needed to restore its state to that which existed when expansion of the scapegoat node was last done.

The total process terminates when a complete tree is found, as determined by IS-COMPLETE, for the algorithm insures that a complete tree is a solution, with the recursion being entirely rewound back to CBS-SYNTHESIZE and a solution tree returned as a value. The total process may also fail to find a solution because of inconsistent constraints and/or synthesis grammar, algorithmic failings, or violating some overall computational limit. Such events will be recorded.

We should note here that the search method used is largely determined by the manner in which nodes are selected for expansion (as embodied in the NEXT-NODE-TO-EXPAND-OF function); it may be depth-first, breadth-first, or some mixed strategy. This choice and many other design considerations are discussed in detail in what follows. However, before treating these matters at length, we complete the description of the essentially utility functions.

The IS-COMPLETE function is given a node and is to return true if the argument node is a root-node and the tree of which it is the root is complete (i.e., a tree devoid of unexpanded nodes), or false otherwise. In the interest of computational efficiency, it is expected that this function will be implemented so that it need not actually scan the given subtree for completeness. If unexpanded nodes are held in a stack, then a node that is a root node is complete if the stack is empty. A similar result is accomplished by

maintaining a count of yet unexpanded nodes; EXPAND will augment this number each time it adds an unexpanded node to the partial tree and reduce it for the node being expanded, while CONTRACT will reduce the count whenever it removes unexpanded nodes. In either form of implementation, the question of completeness is answered directly and need not be computed.

The EXPAND function is given an unexpanded node (which belongs to a particular syntax class) and a case of the synthesis grammar that applies to that node class. From this, it extends the current unexpanded leaf node to nodes for the given case production of the grammar. That is, this is the tree building function. In addition to its primary tree building role, EXPAND performs subsidiary bookkeeping, related to the state of the tree, for the IS-COMPLETE and EXCEEDS-LIMITS functions (see the descriptions of those other functions for an explanation of that bookkeeping).

The CONTRACT function is the inverse of EXPAND; the role of CONTRACT is to return the partial tree to the state it enjoyed just before the matching EXPAND was performed. CONTRACT is called whenever backup is being conducted, and each call will restore one node, the argument node, to its unexpanded state. CONTRACT also performs a subsidiary bookkeeping role for IS-COMPLETE and EXCEEDS-LIMITS (as described in the descriptions of those functions).

The UNPROPAGATE-CONSTRAINTS-FROM function is activated in tandem with (i.e., just before) CONTRACT as part of the backup process. Its effect is to remove any constraints that had been propagated to other nodes from the argument node (by PROPAGATE-CONSTRAINTS-FROM). This function is thus an inverse of PROPAGATE-CONSTRAINTS-FROM.

Node Expansion Ordering

To make the Synthesizer search procedures computationally reasonable, it is necessary for the NEXT-NODE-TO-EXPAND-OF function to make good guesses when choosing the next node to expand and/or to avoid bad ones; good decisions can lead quickly to a solution, while bad decisions may not be capable of leading to a solution at all or may lead to a solution only after excessive computational effort has been expended. Thus, the nature of the problem is such

that much of the ensuing discussion is presented from the negative position of avoiding bad, rather than doing good. If enough bad decisions can be avoided, then the process will be able to stumble upon a solution within the computational limits imposed. It should be obvious that the method used for improving the selection process must then be one that is computationally more efficient than the simpler method that is being replaced; that is, care must be exercised to insure that legitimate optimizations are being provided.

A very simple expansion algorithm is to expand nodes in a strict left-to-right order; that is, the leftmost unexpanded node of the partial derivation tree is always the one selected next for expansion. Such an algorithm falls into the depth-first category. This algorithm has several attractive attributes; it is very easy to understand, it can easily be implemented efficiently, it generates the tree in a prefix order that corresponds to the generally expected node order called for in SEMANOL sequence expressions, and the theoretical properties of this method have been frequently studied. Unfortunately, the algorithm was found inefficient in many synthesis examples that were examined. We are thus led to conclude that it is not adequate for a workable Synthesizer.

In seeking better algorithms, we were largely influenced by a few general heuristic principles. The first was the notion of selecting heavily constrained nodes for expansion before more lightly constrained nodes. An argument along this line in [Golomb and Baumert 1965] is justified on the basis of information theory, and leads to the conclusion that "all other things being equal, it is more efficient to make the choice from the set with fewest elements". An intuitive argument for this position can be made by observing that the greater the degree of constraint, the fewer the derivation trees that can be extended from this node and hence need be considered. Thus it is relatively quickly learned whether the heavily constrained node can be part of a solution or not. If not, backup is induced quickly and computational time thus reduced. If this node can be part of a solution, the most constrained part has already been found

and it should be relatively easier to find solutions for the less constrained nodes yet to be expanded (since they have proportionately more satisfying options). How one measures constraint degree in a practical, yet precise, manner is unknown to us (since the rough definitions given here imply that measurement is equivalent to extending all possible subtrees), but the idea can be informally applied.

The second general heuristic principle used in this project recognizes that the nature of context-free grammars and constraints is such that a decision dependency exists. Making one decision can greatly influence various other decision possibilities. The dependencies appear to be substantially hierarchical in structure in that some expansion decisions are more important than others; that is, they act to constrain other expansion decisions. It is this type of situation that is discussed in [Sacerdoti 1977], wherein it is pointed out that some decisions can be delayed and that, indeed, to make the choice early is to follow an inefficient policy. The general idea is that if, for example, decision B depends upon decision A, decision A ought to be made first and so allowed to constrain decision B. The argument for this policy is then much like that of the previous paragraph. The intention is to:

1. make fundamental choices early, and then follow their dependency chains to determine if they can be part of a solution or lead to a contradiction. In particular, this can then lead to quick backup and rejection of incorrect expansion choices.
2. defer less important decisions, waiting until all applicable constraints have been computed.

These principles can be (at least partly) realized in several ways, some of which are described in what follows. As the grammar drives the expansion process, this is done in all cases by augmenting the grammar with information that can be used to guide the selection of nodes for expansion.

One way in which the grammar can be augmented to control the node selection process is by denoting the expansion order of the non-terminal elements appearing on the right side of any production. The selection algorithm contemplated would then select among sibling nodes on the basis of this ordering relationship; it would also forward this relative ordering to descendent nodes whose expansion, within the inherited order, would be governed by the ordering specified for their productions, etc. This method provides a way in which decision dependency can be communicated to the expansion algorithm. A variation of this type of scheme would allow the user to impose an expansion priority order, based on syntax class membership, upon the process. Here the user would associate with each syntactic class a priority, and each step in the expansion process would select the leftmost node from those having the highest priority. In either of these methods, it is quite possible for ordering relationships or syntactic class rankings to be only partially given; thus the task of annotating the grammar can be made a simple task.

Both of these methods have been tried with simple grammars and constraint sets, and both have proven superior to a simple right-to-left ordering method. They are also only a little more difficult to implement than a left-to-right algorithm. For instance, it is possible to implement all three methods discussed with stack methods. For strict left-to-right expansion, unexpanded nodes would be pushed onto a stack from right-to-left for the production case chosen; the topmost (i.e., leftmost) node is then expanded with the consequence that its stack entry is replaced (popped) by inserting (pushing) the unexpanded nodes resulting from the expansion of this topmost node. The ordering method discussed first above varies this only in that the unexpanded nodes are pushed onto the stack to reflect the user ordering rather than in a simple right-to-left order. The priority scheme essentially expands the method to require that a stack be provided for each priority value. Unexpanded nodes are then distributed across these stacks on the basis of their priority and are selected always from the top of the highest priority non-empty stack.

A third possibility was also considered in this project. It would assign expansion condition expressions to syntactic classes; evaluation of this expansion condition to true for a given node makes it a candidate for expansion, and the leftmost node of those having true conditions would then be selected. (Some default rule would be operative if no condition evaluated to true.) Such a scheme allows greater discrimination to be expressed about node expansion order than would the previous two methods; thus a user is able to more fully convey knowledge of dependencies to the Synthesizer program. Conditions need not be generalized for each syntactic class, but can be evaluated with regard to the state of the partial derivation tree presently being generated. This advantage comes at a great cost. Its implementation would be much more difficult than for the other methods presented, and would require the addition of at least one new SEMANOL primitive. It would also be very costly to use since expansion conditions must continually be evaluated, so the efficiency of the method is suspect. And finally, the preparation of expansion conditions, which now would need to be done manually, places a great burden upon the user. We presently would not consider this an attractive choice; we do suggest that the syntactic class priority scheme be used in a Synthesizer implementation.

Case Selection

Once a node for expansion has been chosen, it then is necessary to select the case from the synthesis grammar which is to be used for that node's expansion. In the CBS algorithm discussed here, cases are sequentially selected from a list of possible cases that has been formed by the COMPUTE-POSSIBLE-CASE-LIST function. It is thus the role of COMPUTE-POSSIBLE-CASE-LIST to order the case possibilities as advantageously as possible and, if it can, to eliminate those cases that ought not to be tried. The resulting ordered list of case possibilities is associated with the (current) node argument, and the list is accessible by use of the POSSIBLE-CASE-LIST function.

The ability to make good case selections is obviously very critical to the success of the Synthesizer, although the relative importance of case selection can be reduced if expansion nodes are selected particularly well. That is, node selection and case selection together will largely determine how efficiently test case synthesis is performed; good performance in one area will somewhat obviate the need for good performance in the other. In the following we consider ways in which case selection might be done.

In considering case selection techniques, an effort has been made during project performance to establish general principles upon which heuristics could be based. One principle is based on failure cost estimation, where cost is expressed in terms of computational time, and seeks to estimate a failure cost for each case that might be selected. If we assume that the node for which this analysis is being attempted is part of a tree which can be extended to a solution, it is then possible to apply the CBS algorithm (including some form of case selection) to determine (1) whether a given case is part of a success or failure and (2) how long it takes (i.e., the cost) to determine if a case is a failure. This, of course, is what the CBS algorithm is designed to do; it also means that the failure cost of any selection is constant in a given situation, and that the probability of success or failure is either 0 or 1 for a given case. However, since determining these actual values can hardly reduce computational time, the question is whether a useful estimated failure cost can be computed at a reasonable expense. The estimated failure cost in this case becomes the product of the likelihood of the case failing and the cost of detecting such a failure. The intention would be to order cases such that those of low estimated failure cost would be tried before those with higher estimated costs. It appears that case failure cost estimates might be (1) inferred from the constraints, (2) supplied by the user, or (3) based upon past Synthesizer performance. Past performance estimates may do no more than reveal a systematic bias in the way constraints were being constructed or simply reflect the inherent pattern of the Synthesizer algorithm but, even so, such information could be automatically collected and, because of its ready collectibility, might be found marginally effective.

Otherwise, the cost of finding a case selection failure is presumed to be best estimated from its apparent subsequent expansion complexity. The cost for cases of single elements would be expected to be less than for more lengthy productions; for example, given

```
a : => <b>          case 1
    => <c>  <d>     case 2
```

it might reasonably be assumed that the computation to find a failure in case 1 would be less than for case 2. And certainly cases involving recursion should probably be considered more costly than those without. It thus seems that some use of this idea might be made part of an automated procedure, although how such a procedure might be influenced by the constraints applied for a given run could not be devised by us. Letting the user provide cost estimates is, of course, an accepted way of merging user intelligence with that of a computer program to increase the power of the overall process.

A second principle appropriate to case selection is that of choosing the case that will least constrain subsequent case decisions. For example, if choosing line numbers that must be ordered and are assigned in left-to-right order, then this principle would cause the smallest remaining line number to be selected each time. This principle directly seeks to identify good choices, and so complements failure cost analysis (which seeks to reduce the cost of bad choices); its general idea was also discussed with regard to node expansion order in this report. It should be noted that this approach has been elegantly employed in the crossword puzzle generator described in [Mazlack 1976]. However, there is no simple implementation that will serve the Synthesizer.

Actual case selection design for the Synthesizer considered several possibilities at some length. A very simple design is to select cases in the order in which they are written in the synthesis grammar, and to then arrange the grammar so as to reflect whatever instincts exist relative to the way in which cases ought to be selected. This might be done once,

or repeatedly so as to include information about the dynamic state of the current derivation tree; it might be done manually by the user or automatically as a consequence of Synthesizer experience (or inference, if that becomes possible). The implementation would then become correspondingly simple or complex.

An alternate design investigated was one that is based on look-ahead. The idea behind look-ahead is that (1) trial extensions of the current partial tree would be made for each candidate case, (2) each trial extension would be evaluated, and (3) the cases would be ordered on the basis of that evaluation. Since each production of the synthesis grammar ordinarily has only a few cases, a look-ahead procedure involving a small number of steps should be computationally manageable (e.g., a two step look-ahead process for a grammar uniformly having three cases per production need consider no more than 12 partial extensions). Thus a limited amount of breadth-first analysis is done in order to improve the choice of a case for depth oriented extension.

Now it is always possible that this trial extension process will lead directly to constraints evaluating to #TRUE or #FALSE, and so will itself find solutions or cases that need not be considered further, respectively. However, if this is not so, the trial extensions must be evaluated in heuristic terms, probably on the basis of some estimate of constraint degree. A measure developed for two step look-ahead would estimate constraint degree as the proportion of failures to possible cases at step two; the higher the ratio, the greater the degree of constraint and the less appealing is the parent case. The step one case with the lowest ratio would then be placed at the head of the POSSIBLE-CASE-LIST, the one with the next lowest ratio second in the list, and so on. Since frequently #TRUE or #FALSE values will not be determinable, this look-ahead ordering will often reflect little more than the case ordering of the synthesis grammar. The degree to which this forecasting method will actually improve case selection can only be

determined by experience. It clearly incurs a substantial expense in performing the trial extensions to the partial tree, and a similar expense as the effects of the trial extensions are undone. (In effect, each trial extension is treated in the same way as is a failure in the central CBS algorithm.) It should also be observed that the power of look-ahead will be improved whenever the ability of partial evaluation to detect success and failure is enhanced.

The CBS algorithm performs an initial case ordering and simply cycles through the pre-ordered list until a solution is found or the list is exhausted. It is recognized that this procedure is deficient in that it fails to reconsider case order when a selected case fails. It would be better, as discussed by [Sussman-McDermott 1972] and others, to use the information gained in conducting a failing search for added guidance when selecting the next case. In future designs, this will be given greater attention at the case level. However, the current algorithm does use this type of information when backing up as explained elsewhere in this report. The emphasis is thus placed upon using this information in node selection and only indirectly does it apply to case selection.

In an initial Synthesizer implementation, it is expected that case selection would be done in the same order as the cases are given in the synthesis grammar. At the same time, user attention can be given to the grammar so that it will be (at least partially) organized to expedite good case selection. Subsequent versions can then consider look-ahead methods if the need becomes evident and further analysis of look-ahead argues for its efficiency.

Constraint Propagation and Partial Evaluation

The routines involved in constraint propagation are COMPUTE-PARTIAL-EVAL-RESULTS and PROPAGATE-CONSTRAINTS-FROM. Together they are intended to reduce the computation time required to evaluate constraints, essentially by (1) reducing the conditions to be tested for at each expansion step to only those that are pertinent at that point and (2) avoiding recomputation of values that were fixed by previous decisions. As will be seen, much of this is done by the COMPUTE-PARTIAL-EVAL-RESULTS function. However, this function also implements a search optimization that we call partial evaluation. The advantage of partial evaluation is that it can be applied to incomplete derivation trees, rather than requiring trees to be fully expanded, with the special advantage that partial trees that cannot possibly be extended to solutions can be recognized; this early recognition can thereby be used to avoid useless computation.

COMPUTE-PARTIAL-EVAL-RESULTS successively evaluates each element of the set of constraints that are bound to the argument node it is given. Each of these constraints is a SEMANOL boolean expression. In some cases, all the nodes referred to in the evaluation process will already have been expanded; in these cases the boolean expression can be fully evaluated and so will have the value #TRUE or #FALSE. More commonly, unexpanded nodes will be encountered in the evaluation; in these cases, some parts of the expression cannot be evaluated because of incomplete information. Thus a simple #TRUE or #FALSE cannot be returned; instead, a new boolean expression will be returned that, when evaluated on any partial tree extended from the present partial tree, will be #TRUE if and only if the original node constraint would be #TRUE for the same partial tree. This constraint reduction process will lead to constraints that are more quickly evaluated due to their inclusion of information about case selection that was not embodied in the initial node constraint. The results of constraint evaluation (i.e., #TRUE, #FALSE, or a new boolean constraint) are placed

in a list, accessed by use of PARTIAL-EVAL-RESULTS-LIST, that is associated with the argument node.

An algebra analogy to partial evaluation may help get the basic idea across. Suppose that it is desired to partially evaluate the relation (x+y+z)=6. If x=1, y=2, and z=3, the partial evaluation can complete, returning the answer #TRUE. But suppose x is not yet defined, while y and z are as before. Then the result of partial evaluation is the relational expression x = 1 which may be true or false, depending on the ultimate value of x. One thing certain, however, is that as long as y and z maintain their values of 2 and 3, respectively, (x+y+z)=6 is true if and only if x=1 is true.

We should point out that the boolean constraints being evaluated here may use the full range of SEMANOL operators. Partial evaluation is thus a complex process that must deal with a wide variety of expression possibilities. It does not deal only with expressions composed of boolean operands and sub-expressions. Nevertheless, it is expected that not all SEMANOL operators need be provided in an initial implementation since some are viewed as unlikely to appear in compiler test case constraints.

The process of evaluation for constraints is essentially one of applying SEMANOL functions, constraints or constraint definitions, to nodes of a derivation (i.e., parse) tree. Such functions may, of course, invoke other functions, etc. When all the argument nodes involved in evaluating a constraint are already expanded in the partial tree, evaluation is no different than it is otherwise for SEMANOL boolean expressions, and a value of #TRUE or #FALSE can be determined. However, when unexpanded nodes are encountered in the evaluation process, a function applied to that node does not yet have a defined value. In this situation, partial evaluation will construct a new boolean expression that, when evaluated on any extension of the current partial tree, will be #TRUE if and only if the original constraint bound to the current node is #TRUE for the same extended tree. This new

constraint may refer to different parts of the tree than the constraint from which it is deduced, constitute a substantially different expression than its antecedent, etc. It will be a constraint that can be more quickly evaluated, since it includes information about the current case decision that the earlier constraint could not know. The new constraints are formulated with full understanding of the current partial tree, and so can naturally be simplified. Consider the simple algebraic example given earlier and the constraint that (value (x)+value(y)+value(z))>6 hold at the current-node; partial evaluation would attempt to form the sum, but would be thwarted at the unexpanded node corresponding to x. The value of the sum would thus be 5+ value(x), and a transformed boolean expression 5+value(x)>6 could readily be associated with the x node as a constraint.

Since constraints are not allowed (in this design) to refer to SEMANOL global variables, the values of constraints are always grounded upon the nodes of the parse tree. This means that incomplete evaluation will include references to unexpanded nodes, and that these references will necessarily stand for themselves at that time. Partial evaluation, when it cannot lead to #TRUE or #FALSE, will thus lead to SEMANOL expressions that include SEMANOL constants and operators and one or more functional expressions that apply to unexpanded nodes of the partial tree. As suggested earlier, partial evaluation will follow essentially the same procedure as does the current SEMANOL system, with a natural augmentation for incomplete evaluation being added. The matter of how these mixed expressions can be simplified was not a major topic of investigation in this project since the constraint expressions are not generally expected to be arithmetic in character. If arithmetic simplification should appear to be of benefit to the Synthesizer, the ideas of systems such as Reduce 2 [Hearn 1971] would be evaluated.

It should be observed that SEMANOL operators, such as #SEQUENCE-OF-NODES-IN, that take nodes as arguments, but require complete subtrees for their evaluation, must be treated specially. A general method of treating these

cases is to delay evaluation of #SEQUENCE-OF-NODES-IN until all subnodes are expanded. However, special cases can be handled more efficiently. For instance,

#FOR-ALL y #IN (#SEQUENCE-OF-NODES-IN cn) #IT-IS-TRUE-THAT (B(y)),

where B(y) is a boolean expression, is #FALSE if B(y) is #FALSE at any expanded node, and is unaffected by the existence of remaining unexpanded nodes. In particular, if B(cn) is #FALSE, then the expression is #FALSE; we believe this will often be the case and that a resulting efficiency can be realized. A similar economy is possible when the SEMANOL #THERE-EXISTS operator replaces #FOR-ALL, although then the #TRUE condition is the one that can be detected from an incomplete tree.

There are additional optimizations that have been identified for possible inclusion in a Synthesizer. If the partial evaluator can assume no evaluation side-effects (a valid assumption), it can optimize partial evaluation of the boolean binary operators. For example, suppose it is called upon to partially evaluate

expl #OR exp2.

It may turn out that expl evaluates to #TRUE and that exp2 cannot be fully evaluated because of references to unexpanded nodes. Then, the whole #OR expression should partially evaluate to #TRUE as well. The only possible pitfall of this technique results if exp2 could later evaluate to #ERROR or #UNDEFINED. The constraint generator (human or machine) must guarantee that this cannot happen.

The #THERE-EXISTS operator and similar user-defined #DFs provide an opportunity for partial precomputation which can help the evaluation process. Suppose there is a node x with a node constraint of the form

#THERE-EXISTS y #IN (#SEQUENCE-OF-NODES-IN x) #SUCH THAT (y #IS <abc>)

for some arbitrary syntactic class <abc> . If it is true that only some cases of x can have subnodes of class <abc> , then these are the only cases that can evaluate to #TRUE. A Synthesizer precomputation could be done to discover, for each syntactic class, which syntactic classes can be subnodes. Then the partial evaluator can return #FALSE for any expression of the above form which cannot possibly be satisfied. The optimization represents a concrete realization of the dependency principle described in general terms earlier.

The implementation of a partial evaluation method would borrow heavily from the existing SEMANOL Executer program. The data structures and many of the subroutines could be adapted to use in the Synthesizer with little change. The major changes would be in the main control logic, which would need to deal with arguments that have only been partially evaluated, and in the operator routines that deal with node or sequence of node arguments and/or results. Still, the reliance upon SEMANOL primitives would greatly aid Synthesizer implementation.

The PROPAGATE-CONSTRAINTS-FROM function completes the constraint propagation process begun by COMPUTE-PARTIAL-EVAL-RESULTS. Partial evaluation causes a partial result (#TRUE, #FALSE, or boolean constraint) to be computed for each constraint associated with the current node. The role of PROPAGATE-CONSTRAINTS-FROM is to pass these partial results to unexpanded nodes in the tree, while insuring that satisfaction of the propagated constraints implies satisfaction of the current node constraints. #TRUE partial values are not propagated, and #FALSE values have caused backup to begin as shown in the main CBS algorithm, so some reduction in constraints is taking place. Furthermore, constraints are propagated on the basis that the present case selection is indeed a correct one; thus constraints become more specific, and,consequently, more quickly evaluated.

The propagation algorithm developed in this project can be explained with the following notation. Let cn be the node designator for the current node which, in the CBS algorithm, is the most recently expanded node of the current partial tree, pt. Let c(x) represent a node constraint which is to be propagated from cn, and $B(d_1, d_2, \ldots, d_k)$ be the result of partially evaluating c(x) when bound to cn in pt. Each d_i is a node designator for an unexpanded node in pt ($k \geq 1$ since this is an incompletely evaluated boolean expression). Note that the d_i need not be distinct and that they generally need not designate the subnodes of cn.

The propagation algorithm determines the first node of (d_i) that is to be expanded by the CBS algorithm, call it d_p. (This clairvoyance can be achieved in several ways, as discussed shortly). It then converts $B(d_1, d_2, \ldots, d_k)$ into a node constraint upon d_p by substituting for each d_i a SEMANOL node-valued expression, with free variable x, which evaluates to the node designated by d_i when x is bound to d_p. Such an expression can always be easily produced since d_i and d_p both designate nodes in some fixed subtree of pt (and SEMANOL provides tree walking operators). The node constraint thereby formed is then associated with node d_p. This process guarantees that the propagated constraint evaluates to #TRUE at d_p if and only if c(x) evaluates to #TRUE at cn for any extension of pt.

The requirement that d_p, an element of an arbitrary set of nodes, can be predicted can be met in different ways depending upon the complexity of the node selection algorithm being used. For instance:

1. If a strict left-to-right expansion order is used, then the leftmost node of any set of unexpanded nodes will be the first one of the set to be selected for expansion.

2. If one of the priority schemes discussed is used, then there is still an algorithmic method, certainly more complicated than the one above, that can be used. In either of these two cases, all that is necessary is that the algorithm for node selection be known to the constraint propagator as well as the node selector.

3. If a more sophisticated expansion order scheme is used, it may become necessary to propagate the constraints to all nodes of the set, and then withdraw the constraints once a first member is selected. Prediction is then replaced by bookkeeping, but the validity of the method is unchanged.

Search Limiting

The CBS-EXPAND-NEXT-NODE function activates the EXCEEDS-LIMITS function each time a new case expansion is tried. EXCEEDS-LIMITS returns true if the partial tree argument exceeds some size criteria, or false if the partial tree is within bounds. The purpose of EXCEEDS-LIMITS is to make the search space finite in such a way that termination may be proved in theory and insured in practice. The way of finding bad expansion paths described so far is through the partial evaluation of constraints that yield a #FALSE result; the partial tree upon which this occurs is then known incapable of being extended to a solution. Ordinarily, constraints do not deal with the size of the derivation tree (even indirectly). Thus recursive loops may be entered, for instance, that will not terminate within a reasonable computational time (or maybe not at all). The EXCEEDS-LIMITS function seeks to impose simply determined heuristics that can overcome unproductive search patterns; in effect, it introduces new constraints that are directed toward the computational process itself rather than directly with the nature of the desired output.

A simple limiting test can be based upon the number of nodes, expanded and non-expanded, that exist in the partial tree. If the node count exceeds some specific limit, then this path is abandoned and backup induced. This method rests on the fact that a program to be generated is generally meant to be a concise test of some language feature, and that the number of nodes in a derivation tree is reasonably related to a test program's size. So if a partial tree is being generated in a moderately economical way, the process is allowed to continue; otherwise, that path will be terminated.

A variation, or augmentation, of this form of limit is one based on tree depth. In this test, the distance of the leaf nodes from the root node is determined and compared with a given limit value. The advantage of this simple depth test is that it will quickly detect a left or right recursive expansion loop that seems to be running away. A refinement of the simple depth limit test would permit specification of the nesting depth to be permitted for any given syntactic class. In this case, backup would occur whenever a syntactic class was repeated more than the specified number of times in a derivation chain reaching from a leaf node back to the root node. The advantage of this form of limit test is that it is a more precise way of detecting recursion problems.

The tests just described are global tests since they apply to the root node and the entire partial tree then existing. The specification of limit values in these cases is easily done since global values are being given. Somewhat more sophisticated versions of the node number test and of the simple depth test can be provided by allowing limit values of either kind to be associated with a specific syntactic class rather than just with the root node. Thus the generation of certain subtrees could be readily controlled. At the expense of more detailed specification, a greater selectivity is realized.

The implementation of EXCEEDS-LIMITS is expected to be one that relies upon an incremental computation of node or depth counts, rather than upon a complete tree scan each time EXCEEDS-LIMITS is called (which is fairly often). The incremental computations will be performed by the EXPAND and CONTRACT functions. As nodes are added to the tree, by EXPAND, the global node count will be correspondingly increased; when nodes are removed, by CONTRACT, the count is decreased. A running node count is thereby easily maintained. A similar procedure can be implemented for the simple depth limit test, except that a depth count will be associated with each node. As nodes are added, EXPAND will compute their depth, which is simply one greater than the depth of the parent node from which they were expanded.

CONTRACT need do nothing to depth counts (except erase them for nodes that are removed from the partial tree). Other forms of limit testing can be treated in analogous ways. In this type of implementation, EXCEEDS-LIMITS need only test a global node count or the depth of a descendent of the current node argument (since the only depth related change since the last test has resulted from an expansion of the current node).

This type of search limiting is especially powerful since it is easily specified by a user and can be performed very efficiently. It thus would be part of any Synthesizer implementation and, given the depth-first bias of the design presented here, we would expect a simple depth test to be part of the initial implementation.

Backup

The CBS algorithm, after selecting a node, will successively attempt to extend the current partial tree into a solution by trying each case possibility for the selected node. However, if no case leads to a solution, then the CBS algorithm will return to a prior decision point in order to try some other path. This process of retreat is called backup. Assuming a consistent grammar and constraint set, a detected failure at some node means that a decision made earlier in the construction of the current partial tree was incorrect. We refer to the node at which the most recent case decision was made that, if remade, would allow the tree to be extended to a complete solution as the "culprit". It is considered to be the point at which the bad decision was made. A "scapegoat" is then the term used to denote the Synthesizer's guess as to the culprit. These terms, and many ideas, have been borrowed from [Stallman and Sussman 1976]. The objective of backup is to find scapegoats that are culprits, or failing that (as we shall), scapegoats that are good guesses as to the culprit.

CBS-EXPAND-NEXT-NODE invokes SCAPEGOAT-NODE to select a scapegoat when the need for backup is recognized. SCAPEGOAT-NODE returns a node that is a subnode of the current incomplete partial tree. CBS-EXPAND-NEXT-NODE returns this scapegoat, as its value, to another instantiation of the CBS-EXPAND-NEXT-NODE routine. The fact that the value returned is not the root of a complete tree is used to identify this result as a scapegoat for backup. The result is then passed through an unwinding series of recursive invocations of CBS-EXPAND-NEXT-NODE until the instantiation is reached for which the scapegoat is the current node. In this unwinding, the partial tree is restored to its prior condition at each step through the use of CONTRACT and UNPROPAGATE-CONSTRAINTS-FROM. The net result is that an alternate expansion can then be attempted and some set of search possibilities can be discarded.

One SCAPEGOAT-NODE algorithm possibility would involve simply finding, and returning as its value, the node most recently expanded prior to the node from which backup is being made. While this requires that node expansion order be maintained for any partial derivation tree, implementation is obviously easily done. The effect of applying this method is always to backup one level, and then to try the next case at that level. Such a procedure will work, in that it will not miss any possible solutions, but its inelegance and obvious inefficiency in many cases argue for efforts to find a more promising way.

The approach that follows is based on [Stallman and Sussman 1976] and depends upon decision dependency. Since backup is a consequence of one or more node constraints evaluating to #FALSE upon each case extension to the node in question, all nodes at which a change of case could alter the truth value of any #FALSE node constraint are potential culprits. This set of nodes will be called the complete dependency set, and it will be defined in terms of a one-step dependency set. The one-step dependency set consists of the following unexpanded nodes:

1. The #PARENT-NODE, since a different case solution at the parent node might remove the need for the subject (syntactic class) node altogether.

2. The nodes that have propagated constraints to the subject node, since a different case selection at these nodes can change or eliminate the constraints placed on the subject node.

3. The nodes that are referenced in partial evaluation of node constraints for all possible case extensions (where the partial evaluation context agrees with that obtained when expanding the cases for this node). Different choices at any of these nodes could affect the evaluation of constraints, and thereby change a #FALSE result to #TRUE because of different operand values.

The one-step dependency set can be seen to include all nodes that had an immediate effect upon constraint evaluation results, and where changes could lead to a different consequence for the subject node. However, these immediate nodes are shaped by the choices made at their one-step dependency set nodes which, in turn, means that the subject node is also affected by case selections made at these once removed nodes, etc. So the subject node is possibly influenced by any member of this widening circle of one-step dependency sets of nodes. This wide circle of nodes is the complete dependency set and is easily computable recursively from the one-step dependency set for a given node and the complete dependency sets for all nodes in that one-step dependency set (i.e., it is the union of these sets). Thus all possible culprit nodes are identified in this process. As required, each node is one that was expanded before the subject node; they all correspond to points at which decisions were made before expansion was attempted at this node. The problem then becomes one of choosing well from among these antecedent nodes the node at which to resume expansion for a different case.

The nodes in the total dependency set can be ordered in terms of their relative position in the node expansion order (which is always known for any partial tree). The scapegoat search can then be conducted in inverse order; i.e., from recent decisions backwards to earlier ones. If any expanded nodes in the expansion order can be bypassed (i.e., are between the subject node and the scapegoat in the expansion order), backing up past them will obviously save time. However, backing up past the most recently expanded member of the total dependency set requires determining that other decisions at this most recent decision point will not change constraint evaluation at the subject node from #FALSE. If there are no cases still to be tried at this most recent decision node, it is certainly safe to bypass it directly. Otherwise, we have no heuristic to propose that would circumvent the simple procedure of then trying the next case. Caution must be exercised in backup to avoid discarding possible solutions. If a node is backed over, the CBS algorithm will no longer attempt alternate case expansions for the backed over node; this means a set of partial derivation trees will no longer be formed and tested. One does not wish to have a solution thrown out in this way.

We recommend selecting the most recently expanded node in the total dependency set for the scapegoat. Efforts to make this work efficiently should be directed at ways in which to reduce the size of the total dependency set. Improved forms of constraint propagation would clearly affect this, especially if constraints can be localized in application. It may also be possible to optimize the process that identifies nodes upon which evaluation depends so that some nodes can be ignored for which dependency does not exist (despite the reference). For instance, case independent conditions, such as x #IS <class>, may permit the referenced node to be excluded from the total dependency set. This is an area that needs further investigation.

Finding a scapegoat when backup is induced by exceeding some form of limit presents a different problem, and so was not discussed in the foregoing. Limits are computational constraints imposed to insure algorithmic termination.

The dependency set in this situation consists of those previously expanded nodes corresponding to case selections that affect the limit being used. If the limit is simply upon the size of the partial tree, than any node with case options left that would reduce the size of the tree would be a possible culprit. Similar dependency sets for the other limit measures mentioned in this report could be constructed when needed. While backup in these cases is important in a production system, our interest in any prototype Synthesizer implementation would be upon determining the reason for reaching the limit, which ordinarily would be unexpected, and deciding how that could be avoided in the future. The development of scapegoat selection heuristics can then be based upon that experience.

This backup procedure results in returning to a previous decision point, at the scapegoat node, and then attempting to extend a partial tree downward into a solution. It may happen, and frequently will, that the "new" tree generated after backup will be (largely) identical with the one just erased during backup. This can obviously happen if the culprit node was one that changed the value of a constraint reference so that #TRUE now results rather than #FALSE on constraint evaluation. It would be computationally beneficial if the information gained in the first expansion trial could be used in subsequent expansion efforts; that is, historical precedents could be retained and failure history not repeated. This problem is called "rediscovering America" in [Berliner 1973].

It appears that the Synthesizer could be embellished so that it would use historical results to influence tree expansion after backup. The method contemplated would essentially remember the tree that existed during backup, and then bias the subsequent re-expansion in the direction of generating the same tree as was erased. This would be accomplished by having the CONTRACT function note the cases used for expanding any immediate descendent nodes, the nodes being erased, and moving these cases' indicators to the trimmed node that remains. As the recursive case decision structure is rewound back to the

culprit node, the cases used previously are propagated backward to their ancestor nodes. It then would be possible to directly regenerate the partial tree that was erased. However, since the use of a different case at the culprit node may change the propagated constraints and/or dependent values, it is unknown whether or not the prior expansion is still valid at all. Thus the expansion steps must be performed as they otherwise would be. The improvement proposed is to use the remembered case selections when forming the POSSIBLE-CASE-LIST at each node. The intention is to give priority to the case selection used previously, provided it continues to be a valid possibility, and so place it first in the list. If any remembered case is found to fail during tree regeneration, the process of giving preference to previous case selections should probably be abandoned. But until that occurs, the expansion process would repeat its past "success". The improvement is thus largely implemented in the way in which cases are ordered for expansion. The use of a method following this outline appears to be computationally reasonable and thus worthy of more detailed consideration, as is further analysis to determine the probable efficiency of this heuristic for test case generation.

The Synthesizer Interface

The importance of the user interface to the acceptance of any computer program is generally recognized. In the case of AI programs, and certainly the Synthesizer, the quality of the user interface is especially vital to achieving useful results since the program and user are so closely coupled in the process. The interface discussed here is one meant for use in the development of the Synthesizer; it is thus an interface designed for a user who is very familiar with the internal structure of the Synthesizer. However, elements of this developmental interface would be retained, probably in somewhat altered form, in any program that passed into a more public domain. Naturally, the user interface can itself be expected to evolve as the Synthesizer design is developed and becomes more sophisticated; the interface given here is to be taken as a starting point. It should be noted that the interface specifications assume that the Synthesizer will be developed upon a Multics System (since past SEMANOL programs were developed for Multics), but they could obviously be modified to suit other implementation environments.

It is expected that Synthesizer user commands will be modeled after those previously implemented for the SEMANOL Executer program. Thus each command will be packaged as a separate program and communication will be carried on through shared Fortran COMMON blocks.

The extensive interface discussed here is obviously a substantial body of code to implement, most notably with regard to user trace and control facilities. It is felt that such an extensive interface will be needed to effectively support Synthesizer program development testing and, perhaps even more importantly, to then evaluate the effectiveness of the heuristics being used. A continuing process of heuristic improvement is to be expected. In a sense, this means an artificial intelligence program should be able to "explain its reasoning" to the user [Stallman and Sussman 1976]. In the case of the Synthesizer, this means that the system should be able to list (any or all of) its decision points together with relevant information (constraints and heuristics) that led to the decision made. This allows the user to debug the program and gain confidence in the reasoning process. It also helps determine which constraint variations make the synthesis process more effective. There are two ways in which the Synthesizer will be able to explain its reasoning:

(1) It will have a dynamic trace mode which provides background information at each decision point.

(2) It will have an interrogation facility which allows the user to determine the computational state at selected points in the processing.

The specific commands to be provided are these:

1. Initialize-Synthesizer will initialize the Synthesizer program and COMMON blocks used for communication, and will then cause loading of the synthesis grammar and constraint definitions from which test programs are to be generated. This command need be executed only once during a session, but must be given before any other command.

2. Synthesize will cause root node constraints to be loaded and Synthesizer program execution to begin. It as also expected that this command can be used to cause incremental changes to the synthesis grammar and/or constraint definitions to likewise be loaded.

3. Trace commands, similar to those that are used with the SEMANOL(76) Executer to allow semantic definitions to be traced, will allow the user to observe Synthesizer program behavior. These commands will permit selective trace control to be applied to syntax class names of the synthesis grammar, thus making the trace output especially meaningful. The trace of the proposed constrained backtrack synthesis algorithm would include the following:

(1) At each call of CBS-EXPAND-NEXT-NODE:

- Print the invocation level number.
- Print the partial tree input argument, indicating what node constraints are associated with each unexpanded node.
- Print a pointer to the node selected for expansion (the current-node).

(2) For each case that is rejected, print the reason (e.g., that the search limit was exceeded or the node constraint which evaluates to #FALSE).

(3) At each return from CBS-EXPAND-NEXT-NODE:

- Print the level number.
- Print the result (either a complete tree or a scapegoat node identifier).
- In the backup case, indicate which intelligent backup heuristics were used to select the scapegoat node.

4. Breakpoint commands will give the user the ability to interrupt Synthesizer program execution in order to determine the program's current internal state.

It is expected that the user will have the ability to specify conditional breakpoints applicable upon entry to, and return from, CBS-EXPAND-NEXT-NODE. Generally, it appears that breakpoints can be established at any point at which tracing output can be generated. While breakpoints can be set and reset at will, no ability to change the computational state otherwise will be given the user; breakpoints are meant to largely yield an opportunity to use the display commands.

5. Display commands will be available to the user at Synthesizer program termination or at a program break. At either point, the user may cause to be displayed such things as:

(1) The current level number.

(2) The current partial tree, or parts of it, including the constraints associated with each expanded node.

(3) The current node, POSSIBLE-CASE-LIST, PARTIAL-EVAL-RESULT-LIST, and other data structures that are fundamental to the operation of the Synthesizer program.

(4) The past history of expanded nodes, showing any cases that were rejected and the reasons for their rejection. This facility may pose difficult implementation problems, so its inclusion is less certain than that of the other display options.

It is felt that this type of interface will support Synthesizer program development effectively.

4.2 The Analyzer

The object of each execution of the Analyzer is to produce syntactic root node constraints, supporting constraint definitions, and a synthesis grammar; from these, the Synthesizer will produce a specific test program. While these produced elements are entirely syntactic in nature, they must be adequate to cause the Synthesizer to generate semantically legitimate programs. In addition, some test programs will be constructed that rely on input during execution; for these programs, the Analyzer will also produce constraints that must be satisfied by the input to the programs. However, these constraints are not presently meant for the Synthesizer, since the Synthesizer does not generate test program input data but, instead, must be used in manual input data preparation.

The Analyzer is given, as input, the criterion which the final test program is to satisfy. This criterion is phrased in terms of the SEMANOL specification of the object language, which is the other input given to the Analyzer. For example, one plausible criterion would be "exercise case 1 of goto-successor". That is, when executing the test program using the SEMANOL specification and the SEMANOL programming system, the flow of control must go through the first case of the SEMANOL semantic definition "goto-successor". This is a semantic criterion; it is defined in terms of the execution history of the desired test program. It is the job of the Analyzer, then, to translate such semantic conditions to purely syntactic conditions.

One technique to be used depends upon the concept of subsetting the set of semantic modules (i.e., #DF's, #PROC-DF's, and the #CONTROL-COMMANDS section) of the SEMANOL metaprogram. This activity would be performed by the user of the Analyzer, and would (by convention) obey a proper nesting rule; if any two subsets have any module in common, then all the modules of one are also in the other. Then the selected subsets can be used to form a relevant part of the metaprogram for a particular Analyzer run. In this scheme, the relevant part, rather than the entire SEMANOL metaprogram, is

given to the Analyzer. Unlike the full metaprogram, which always has a unique execution entry point, the #CONTROL-COMMANDS section, a relevant part does not necessarily have a unique execution entry point. Thus it is necessary for the user to designate such an entry module. While the Analyzer design is relatively unaffected by whether or not subsetting is provided, subsetting does appear likely to make the Analyzer better able to produce good constraints. It is a helpful technique, especially at this early stage of development.

The design proposed here for the Analyzer was most strongly influenced by the work done on the Synthesizer program. For example, the basic structure of the Analyzer reflects quite faithfully the structure of the Synthesizer. The Analyzer also bears strong resemblance to several other proposed and implemented facilities for program testing. Among these are the SELECT system of SRI [Boyer et al 1975], the EFFIGY system of IBM [King 1976], Howden's DISSECT system [Howden 1977], the ATDG system of TRW for NASA/Johnson Space Center [Hoffman 1976], the RXVP system of General Research Corporation [Miller et al 1975], and Clarke's (unnamed) system [Clarke 1976]. A related approach of interest is reported in [Huang 1975].

The Analyzer design shares with all these systems the notion of path generation. However, a unique scheme was developed here for generating paths covering an incomplete, user specified, list of elements in the (meta) program, where the elements may exist in separate subprogram units which may call each other recursively. Some systems, such as SELECT and EFFIGY, allow paths including recursive invocations of subprograms, but such paths must either be completely determined by the user or exhaustively enumerated (up to some loop limit). Other systems, such as DISSECT, allow automated path completion of partially specified paths but only in the context of single program units. Researchers at the University of Colorado have extended path completion to multiple subprograms, but not recursively callable subprograms [Gabow et al 1976].

The Analyzer design also shares with all of these systems (except ATDG), the idea of developing a set of relations on the input variables sufficient to cause the generated path to be followed by the executing program (in our case, the metaprogram). Along with SELECT, EFFIGY, DISSECT and Clarke's system, the Analyzer relies heavily upon the use of symbolic execution to deduce the appropriate input relations. It was attempted here to extend the domain of the input variables from numeric values (as required in all the above systems) to the highly structured strings which constitute computer programs. The consequences of these extensions upon the Analyzer (especially with regard to symbolic execution) were partially, but not completely, determined.

Some systems, such as SELECT and Clarke's system, share with the Synthesizer (discussed earlier), the notion of resolving the relations on the input variables to produce particular instances of test data. Here, the extension of the domain of interest to include programs had the greatest effect, in that the Synthesizer design differs radically from the solvers used by Boyer, et al, and Clarke.

The discussion that follows deals exclusively with that part of the Analyzer algorithm that is devoted to finding a legitimate execution path through the SEMANOL metaprogram. This emphasis largely reflects the nature of the technical difficulties, as well as the focus of our research; it also reflects our belief that the other parts of the total Analyzer design are adequately presented in Section 3 of this report. This discussion assumes that the synthesis grammar is essentially identical with the #CONTEXT-FREE-SYNTAX section of a given SEMANOL metaprogram; no consideration is given to transformations that might be made to help the Synthesizer.

The Analyzer Algorithm for Path Generation

The proposed path finding algorithm for the Analyzer is essentially recursive and similar in many ways to that developed for the Synthesizer.

The top level control procedure would guide this process by:

1. Constructing a partial path that consists only of an invocation node for the entry module.

2. Associating the path criterion, in the form of path coverage constraints, with that node.

3. Initializing the symbolic state.

4. Calling a recursive procedure, EXTEND-PARTIAL-PATH, to initiate a step-by-step effort to construct a suitable execution path from the current partial path.

5. Receiving a return of control from EXTEND-PARTIAL-PATH only after a successful execution path has been found or the procedure gives up in failure.

It is thus left to EXTEND-PARTIAL-PATH to find a solution.

A description of EXTEND-PARTIAL-PATH is given in Figure 8. It can be seen there that the algorithm first tests to determine if the target path it has been given is complete or not; for if the path is complete, then a solution has just been found and the returned COMPLETE PATH FOUND value will cause the recursion to be unwound. A complete path is defined to be a path in which all module invocations have been expanded. If the path is not yet complete, an (unexpanded) module invocation node is selected, and a path through that module (which is so far untried) is selected and used to expand the selected module invocation. The path to be used in the expansion is tested to see that it meets a limit constraint. The node expansion step is accompanied by the rewriting of path constraints and by the updating of the symbolic execution state.

```
*****************************************************************************************
*                                                                                       *
*  1    IF THE PARTIAL PATH IS COMPLETE                                                 *
*  2        RETURN WITH VALUE: "COMPLETE PATH FOUND".                                   *
*  3    ENDIF                                                                           *
*  4                                                                                    *
*  5    CHOOSE THE NEXT MODULE INVOCATION NODE, N, TO REPLACE; CALL THE MODULE IT INVOKES M(N).  *
*  6                                                                                    *
*  7    DO WHILE ANY THUS FAR UNCHECKED PATHS IN M(N) REMAIN WHICH WOULD NOT CAUSE THE PARTIAL  /*
*           PATH TO EXCEED THE LIMIT:                                                    *
*  8        CHOOSE THE NEXT MODULE PATH, P, IN M(N).                                    *
*  9                                                                                    *
* 10        (NODE EXPANSION) REPLACE N WITH P IN THE PARTIAL PATH, REWRITE THE PATH COVERAGE  /*
*                CONSTRAINT TO REFLECT THE REPLACEMENT, AND AUGMENT THE INCREMENTAL SYMBOLIC  /*
*                EXECUTION STATE.                                                        *
* 11                                                                                    *
* 12        IF THE REWRITTEN PATH COVERAGE CONSTRAINT IS FALSE, OR THE SYMBOLIC EXECUTION STATE  /*
*                IS DEMONSTRABLY INCONSISTENT                                            *
* 13            (REVOKE NODE EXPANSION) REPLACE P WITH N IN THE PARTIAL PATH, RESTORE THE PRIOR  /*
*                    PATH COVERAGE CONSTRAINT, AND RESTORE THE PRIOR SYMBOLIC EXECUTION STATE.  *
* 14        ELSE                                                                        *
* 15            RECURSIVELY CALL EXTEND-PARTIAL-PATH.                                   *
* 16                                                                                    *
* 17            IF THE RESULT OF THE RECURSIVE CALL WAS "COMPLETE PATH FOUND"           *
* 18                RETURN THIS CALL WITH VALUE: "COMPLETE PATH FOUND".                 *
* 19            ELSE                                                                    *
* 20                (REVOKE NODE EXPANSION) REPLACE P WITH N IN THE PARTIAL PATH, RESTORE THE  /*
*                        PRIOR COVERAGE CONSTRAINT, AND RESTORE THE PRIOR SYMBOLIC EXECUTION  /*
*                        STATE.                                                          *
* 21            ENDIF                                                                   *
* 22        ENDIF                                                                       *
* 23    ENDDO                                                                           *
* 24                                                                                    *
* 25    RETURN WITH VALUE: "BACKUP REQUIRED", SINCE ALL ATTEMPTS TO EXPAND N HAVE FAILED.  *
*                                                                                       *
*****************************************************************************************
```

Figure 8: EXTEND-PARTIAL-PATH Procedure

This expansion is then tested to see that it meets the path coverage constraints and that it does not create an inconsistent symbolic state. If either condition is violated, this particular expansion is undone and the next successive path option is tried in its stead. If all path expansion options for this module have been tried or the limit exceeded, backup to a prior decision point (i.e., the prior invocation of EXTEND-PARTIAL-PATH) is made and another subpath tried. The various elements of this procedure are considered in more detail in what follows.

It should be noted that a limit on path length is imposed in order that the algorithm can be assured of termination (we also are only interested in relatively short execution paths for compiler test programs). This may exclude some paths from discovery but, if the limit is adequately high, should not prevent the algorithm from finding acceptable solution paths.

Path Coverage Constraints

Path coverage constraints are a way of relating the original path criterion to the current partial path. Whenever a coverage constraint resolves to true, the original path criterion has been satisfied by the partial path, and whatever remains to be expanded to complete the path is unconstrained by the original path criterion. Whenever a coverage constraint resolves to false, the original path criterion has been proved unsatisfiable for any possible extension of the current partial path. Since the use of path coverage constraints is so vital to the algorithm, they are discussed in detail first.

A partial goal of the Analyzer is to find an execution path that covers designated structural elements of a SEMANOL metaprogram in a given order. This means that a solution path will include nodes that correspond to these structural element goals. The goals do not refer directly to nodes in the execution path, but only to instances of activation of the designated structural elements. Thus the position of a node in the

execution path and the structural element to which it corresponds are both important. These two characteristics of a node will be denoted by a two part identifier in this discussion; this form is (i:t), where i identifies its position in the execution path and t identifies the metaprogram structural element to which the node corresponds. The goals for the Analyzer are thus given as $<t_1, t_2, \ldots, t_n>$; i.e., as a list of structural elements to be activated. A list of path goals applied to a node will be denoted by $<t_1, t_2, \ldots, t_n>$ (i:t); this list of path coverage goals, combined with the single path node to which they apply, is called a <u>primary constraint</u>. A (path coverage constraint) <u>term</u> is then defined to be a conjunction of primary constraints, and a (path coverage constraint) <u>expression</u> is defined to be a disjunction of terms.

The initial path coverage constraint is just the given path criterion applied to the single invocation node that is the initial partial path. It has the meaning that the goal in expanding this node is to provide a path that covers, in order, nodes labeled with the designated structural element identifiers. Any module path expansion for this node which precludes attainment of that goal will be considered inconsistent with that coverage constraint, and that path will be discarded as soon as the inconsistency is detected.

When an invocation node is replaced by its newly chosen module path, the coverage constraint must be rewritten to eliminate any primary constraints which apply to the invocation node, replacing each of them with a coverage constraint expression containing new primary constraints applied to nodes of the newly expanded subpath. Thus a particular allocation of $<a_1, a_2, \ldots, a_j>$ (k:ak) would be written $<a_1>$ (k1:ak1) <u>and</u> $<a_2>$ (k2:ak2) <u>and</u>...<u>and</u> $<a_j>$ (kj:akj) where the (ki:aki) are nodes, in order, in the newly chosen module path for invocation node (k:ak).

This rewriting must have the effect of preserving the requirements of the original coverage constraint. Therefore, the replacement expression for each primary constraint must, in effect, represent all the possible allocations of the goals of the replaced node to the newly created subpath nodes.

Some new subexpressions created when rewriting takes place may be simplified by observing that they are resolvable to either true or false. In particular, primary constraints of the form <ak> (k:ak) always resolve to true, since their meaning is that a node labelled ak must be visited at least once on the subpath consisting of this node labelled ak. Some primary constraints, <ai> (k:ak), resolve to false because they have one of two properties: (1) (k:ak) is a terminal node (that is, not a module invocation node) and ak is not the same label as ai or (2) (k:ak) is a non-terminal module invocation node and the module it involves can be shown, via static call graph analysis, to be unable to reach the module containing the node labelled ai. In the more general case, <...ai...> (k:ak) is false by case (2) if the module invoked by (k:ak) cannot statically reach all of the modules containing the ai (all of which are distinct from that of ak). Finally some primary constraints are unresolvable. These have the form <...ai...> (k:ak), where the module invoked by (k:ak) can statically reach (or is identically) each of the modules containing the ai, but it is not yet known whether all the subgoals can be reached in their proper order.

Primary constraints which are tautologically true are said to be <u>consistent</u>; those which are tautologically false are <u>inconsistent</u>; those which are unresolved are <u>possibly consistent</u>. Thus each primary constraint reflects in its form the allocation of (sub) goals to a particular partial path node, and in its resolution the prognosis for the node's ability to satisfy the goal. Note that it will sometimes be possible to simplify the result expressions by using tautologies such as true <u>and</u> exp ≡ exp, false <u>or</u> exp ≡ exp, etc.

If a node expansion results in a rewritten coverage constraint expression which resolves to false, the module path just expanded is inconsistent with the prior node coverage constraint, and hence with the initial coverage constraint. This only occurs when there are no consistent or possibly consistent allocations for the new subpath; furthermore, the failure to find consistent allocations renders the entire new coverage constraint expression inconsistent. In this case, the expansion of the subpath must be revoked and a new one tried. If a node expansion results in more than one possibly consistent allocation of goals, the new coverage constraint will be expressed as a disjunction of terms, each term being a conjunction of primary constraints.

Since SEMANOL allows recursion among metaprogram modules, the size of the complete coverage constraint can grow monstrously (as is common among symbolic processors [Moses 1971]). Fortunately, there is a simple heuristic which can cut this growth. The heuristic deletes any new term obtained from the old by replacement of one argument node with another of the same type (i.e., with the same label). The effect of this heuristic is to ignore recursions that make no progress toward satisfying the coverage constraint. An added benefit of applying this heuristic is that the reduced coverage constraint is also more useful to the node selection and path selection heuristics than is the original version.

In summary, each revised coverage constraint is kept in a disjunctive normal form, with each term of the disjunction corresponding to a particular allocation of node coverage goals which, as a whole, is possibly consistent with the current partial path (since its primary constraints are). Those terms which have become inconsistent at any point are deleted by the simplifications used; such terms are rendered inconsistent by the inconsistency of (one of) their primary constraints. Whenever a term becomes consistent (because all its primary constraints have become consistent and thus resolve to true), the term, and hence the entire node coverage constraint, resolves to true and no further coverage requirements are posed on yet-to-be-made partial path expansions. Whenever all terms become inconsistent ,

the entire coverage constraint simplifies to false; this is sufficient to backup the partial path construction process.

Node Selection

In the EXTEND-PARTIAL-PATH procedure, a selection is made of the next invocation node to be expanded. Associated with that node is a module flowgraph, encoding implicitly the possible subpaths through the invoked module. Once the invocation node has been selected, an attempt will be made to find some subpath which does not negate the path coverage constraint. If no such subpath can be found, both the invocation node choice and the prior subpath choice (for the prior invocation node) are rejected; they have been proved incapable of leading to a satisfactory solution path. To indicate this rejection, the current invocation of the EXTEND-PARTIAL-PATH procedure returns with the signal BACKUP REQUIRED.

There are several principles, outlined in the discussion of node selection for the Synthesizer, which are generally applicable to backtracking algorithms such as this one. The most general principle is to make as few mistakes (i.e., choices requiring later backups) as possible. A second is to discover such errors as soon as possible so as to minimize the cost of backup.

One of the tools we have to help avoid mistakes, and to discover mistakes as soon as possible, is the path coverage constraint. At any point in partial path expansion, the (rewritten) coverage constraint is an expression of the aspects of the original node coverage criterion that have not yet been resolved by prior decisions (which have produced the current partial path). Thus the coverage constraints can provide hints as to the best future decisions to make in either of the two primary states of the algorithm: (1) the incomplete success state, in which the partial path can be extended to a complete path satisfying the original criterion, or (2)

the undetected failure state, in which the partial path cannot be satisfactorily extended, but that fact has not yet been proved.

The heuristics developed for node selection are given with regard to the degree of a primary constraint or a coverage constraint term. The degree of a primary constraint is defined to be the number of node labels in the goal list portion of the primary constraint, while the degree of a constraint term is the maximum of the degrees of its primary constraints. Thus the degree of term t_1:[<a,b> (n_1) and <c> (n_3)] is 2, whereas the degree of term t_2: [<a> (n_1) and <b> (n_2) and <c> (n_3)] is 1. The notion of degree can be combined with another measure of term complexity, the number of primary constraints in the term, to define a partial ordering of terms. Of particular interest is the notion of a minimal-degree term; a term of a coverage constraint expression is of minimal-degree if there is no other term of lower degree, and if there is no other term of the same degree which has fewer primary constraints. Given these definitions, some heuristics for invocation node selection can be presented concisely.

The principal heuristic is to always select an invocation node for expansion which is mentioned somewhere in the path coverage constraint expression (unless, of course, it has already resolved to true). If the algorithm is in the incomplete success state, expanding nodes in the coverage constraint first will help find the successful subpaths through the constrained portion of the partial path, allowing the unconstrained expansion of the remaining parts of the path to proceed with less overhead. The real savings accrues, however, when the algorithm is in the undetected failure state. In this state, expanding nodes extraneous to the coverage constraints is extremely costly for two related reasons: (1) the discovery of the failure is postponed, since the constrained nodes are the only source of detection, and (2) once the discovery is made, all possible expansions of the irrelevant nodes will be attempted before the backup will finally retract the culprit decision. These observations strongly support using path coverage constraints to govern invocation node selection. Also note

that this heuristic interacts with the recursion limiting heuristic described earlier in a way that prohibits non-progressive recursion; this is a valuable benefit of this heuristic.

A particularly useful refinement is given in the following heuristic: select a node from a highest degree primary constraint in a term of minimal-degree. This heuristic is based upon a strategy of reducing the degree of the minimal-degree terms until terms of degree one are obtained. And it does tend to drive a minimal-degree term to degree one, since subgoal allocation to the nodes of the new subpath will tend to cause replacement of the highest degree primary constraint with new primary constraints of lower degree. The advantages of obtaining a degree one term are substantial. In a degree one term, every primary constraint has exactly one goal node label. Since each primary constraint exists if and only if it is possibly consistent, there is a statically determined path from the argument of the primary constraint to an instance of its goal. Thus each goal node has been proved reachable from unexpanded nodes in the partial path. If each such goal node is on a path which also returns properly, the search for a complete satisfactory path will succeed; furthermore, success will be obtained by systematically expanding each of the nodes in the degree one term to a path satisfying its degree one goal.

The probability is very high that each such goal node is on a path which returns properly, and therefore that this heuristic will expedite the successful attainment of a complete satisfactory path. Indeed, the only way this can fail is that every goal node on all possible (complete) subpaths to be (ultimately) generated for the invocation node has the additional property that its subpath terminates in the computation of #ERROR or #STOP and therefore does not return. This is extremely unlikely. Finally, the heuristic also tends to resolve shorter degree one terms before longer ones, since the amount of work required is presumably less for the shorter terms.

Since this heuristic does not completely determine invocation node selection, some additional refinements may be in order. When several nodes appear to be equally desirable, it may be useful to choose a most-constrained node. There are several measures of the extent to which nodes are constrained. One measure is the number of terms in which the node is mentioned; another is the cumulative degree of the primary constraints mentioning the node. The ultimate value of using these refinements is unclear.

Module Path Selection

Once an invocation node has been selected for expansion, successive subpaths through the invoked module are generated as replacements for the selected invocation node. The subpaths used for this replacement are formed from the flow graph for the invoked module. A flow graph for a SEMANOL metaprogram module is a collection of nodes and directed arcs. The nodes reflect the specification of computational events, while the arcs reflect the specification of transitions between these computational events. All functional definitions (#DF's) have a common flow graph structure reflecting each of the possible case selections. Procedural definitions (#PROC-DF's) and the #CONTROL-COMMANDS section have more general structure. Any of these may induce loops in the flow graph because all may contain iterator constructs.

These flow graphs are then used in the generation of subpaths, with the objective of generating subpaths in an order that can lead to expeditious success. Thus the generation should favor subpaths that reduce the degree of coverage constraints, and so liberally distribute the coverage goals of the invocation node to as many subpath nodes as possible. Conversely, subpaths which allow no allocation of subgoals to the new nodes are generally not to be preferred. This property is statically determinable from the call graph and, therefore, is reasonably inexpensive to confirm. Hence a particular heuristic of value is to postpone generating any such subpaths,

and to generate them only if more optimistic subpaths have proven fruitless. Note that these paths cannot be ignored, because the eventual solutions for a particular criterion may all contain such module subpaths somewhere in the complete path. Also note that, in certain cases, the decision to defer such non-progressive cases may produce larger than necessary programs. Finally, this heuristic may be more appropriate for certain relevant parts of the metaprogram than for others. Despite these uncertainties, our initial experimentation suggests that this is a productive heuristic.

The desired generation order can be realized by a modification of a monotonic path generation algorithm that uses conventional breadth-first graph search techniques (e.g., [Nilsson 1971]). The modification is to replace the partial path queue with a priority queue in which non-progressive paths are placed on a low priority portion of the queue rather than being generated immediately. A generalization of this approach can be obtained by more completely ranking the possible paths according to their ability to distribute coverage subgoals among their component nodes. There are several problems here, all having to do with the cost of such generality. First, the set of paths to be ranked may be very large. Second, the ranking procedure may require the use of the subgoal allocation phase of the coverage constraint rewriting algorithm to produce the measures of progressiveness used for the ranking.

For example, one such measure is the maximum degree of the suballocation terms as rewritten from the primary constraint of the selected invocation node. The subpaths would be ranked inversely with respect to this measure; the higher the maximum degree for a subpath, the lower the subpath's priority would be. Another measure is the minimum degree of the suballocation terms; here the subpaths would be ranked directly with respect to the measure. This measure most closely supports the strategy, but is also the most costly to compute. Observe that this is, in effect, a form of lookahead.

Rather than ranking all limit-satisfying subpaths, it is also possible to enumerate subpaths in monotonic order until a subpath of pre-calculated maximum ranking is obtained. This preferred subpath would then be generated and, if it sufficed, none of the longer subpaths would have to be ranked. The subpath acceptance level can also be relaxed to suboptimal, to precipitate shorter ranking lists. Ultimately, the relaxation may reduce this heuristic to the easily computable heuristic of delaying non-progressive subpaths, presented first.

Symbolic Execution

Not only is it necessary that an interpretation path cover the criterion nodes, it is also required that the path reflect the correct execution of a correctly formed object language program. Any path generation algorithm which only considers the coverage requirement will generally fail to produce a path satisfying the legality property.

There are two types of illegality which can arise: *syntactic and semantic*. Some object language programs, although syntactically correct, are semantically erroneous. For example, a BASIC program whose first executable statement is a RETURN statement is syntactically acceptable, even in the context-sensitive sense. However, it is semantically erroneous, since no prior GOSUB statement has been executed to establish the BASIC statement which is to receive control after execution of the RETURN. If the path generation algorithm produces a path implying such a semantic error in the object language program, the algorithm will produce constraints for a test program that may be semantically illegal.

A path which is not syntactically legal cannot lead to the synthesis of a well formed program. Note that context-sensitive syntactic restrictions are generally not at issue here (so long as the metaprogram is well formed). Rather, the issue is one of path consistency. For instance, some computational

events represented in the path may occur only if the first executable statement in the corresponding object language program is an assignment statement, whereas other computational events represented elsewhere in the path may occur only if the first executable statement is a print statement. Since the first executable statement is unique (for a given execution of a particular program, at least), there is a syntactic inconsistency in the path.

Syntactically illegal paths lead to contradictory root node constraints. Therefore, the Synthesizer will be unable to generate test programs for them. Syntactically legal, but semantically illegal, paths lead to consistent root node constraints, and thus well formed test programs can be synthesized. It is possible to organize the system so that the Analyzer is not burdened with the task of producing legal paths, as well as covering paths. In such an organization, illegal paths would ultimately be detected by the Synthesizer (if they were syntactically illegal) or by the instrumented SEMANOL Interpreter (if they were semantically illegal). The detections made would be transformed into revisions of the node coverage criterion, and the Analyzer would try again. The cyclical organization has two problems: (1) it appears to be grossly inefficient and (2) it is by no means clear how to deduce revisions to the criterion from the error detections.

For these reasons, and because we could see how to guarantee consistent paths incrementally, it was decided to design the Analyzer so as to insure consistency. It is recognized that the cost of generating a single path is greatly increased by the methods thereby used.

The Analyzer uses symbolic execution (e.g., [King 1976]) as the basis of excluding illegal paths. The execution of a metaprogram is symbolic if, in place of candidate test program strings, a symbolic value is used as the value of #GIVEN-PROGRAM and, instead of an actual input string, a symbolic value is used as the value of #INPUT. The symbolic values are used in the static mathematical sense to represent some unknown yet fixed value.

The result of metaprogram operations on symbolic data is a new symbolic datum reflecting the transformations performed by the operations. For example, if A is associated with symbolic value "ALPHA" and B with "BETA", then evaluation of A + B produces the symbolic value "ALPHA + BETA". Symbolic evaluation may involve extensive algebraic simplification: for instance, A + B - A might evaluate to "BETA" rather than "ALPHA + BETA - ALPHA."

The results of symbolic execution are recorded in the symbolic execution state that is maintained by the Analyzer. The symbolic execution state contains properties of paths that bear upon their legality, and it is revised as new subpaths are generated. The symbolic execution state thus contains information that is related to conditional decisions implied by the path that is being generated. When a conditional decision node in a flow graph is expanded and made part of the generated subpath, a path choice is necessarily made. The generated arc on the expanded execution path corresponds to evaluating the semantic boolean expression that caused this flow choice to be made; thus its symbolic value expresses a requirement upon the symbolic variables. This requirement, called a partial path condition, is recorded in the symbolic execution state.

When a semantic definition with parameters is invoked, a new unique symbolic value (called a symbolic designator) is associated with each formal parameter; this symbolic designator stands for the value of the formal parameter for the particular invocation. The symbolic value of the actual parameter is bound to the unique symbolic designator of the formal parameter in a step much like assignment. Successive values of metaprogram global variables are also represented by symbolic designators, bound to symbolic values. These relationships are also recorded in the symbolic state.

The symbolic execution state is checked for consistency as new subpaths are generated. That is, if the expansion of the new subpath has created a property of the state which demonstrably cannot hold simultaneously with some other property of the state, then the subpath is rejected, and another must be tried. For example, if the partial path establishes that, at the point of an invocation mode, the current statement is an <assignment-statement>, then any subpath which is expanded for the invoked module must be consistent with that property. For instance, it must not imply that the current statement is a *<print-statement> at the beginning of the subpath.*

The symbolic execution state and its incremental maintenance constitutes a simulation of the operation of the SEMANOL metaprogram upon the implied test case. Since the test case is implied incompletely by any incomplete partial path, such a simulation is incomplete for any such path. The consistency check upon the incremental state is a way of answering the question: could the execution of a completely determined test case really proceed in this way?

The structure of the symbolic execution state is thus closely related to the structure of the states of the metaprogram as it progressively interprets the (goal) test case. There are, however, major differences:

1. The actual states of the metaprogram will proceed in linear order of execution time, and each state records every aspect of prior states needed for future transitions; thus prior states are discarded. The symbolic execution state is not maintained in the order of execution time. Certain parts of the path may well be completely determined while temporal predecessor portions remain unexpanded. Therefore all temporally prior substates need to be kept as they exist, and the current substate for the just expanded subpath may depend upon incompletely elaborated prior substates.

2. The actual states of the metaprogram will record only information pertaining to the precise structure of the (goal) test program (and its input). Since the goal test program is generally not completely determined during path generation, a symbolic representation of it must be used.

3. Consistency of the actual execution state is guaranteed; consistency for the symbolic execution state must be demonstrated.

The symbolic execution state actually records the successive values to be assumed by the global variables in the metaprogram, and the relationships (recorded in partial path conditions) among those values as implied by the decision arcs explicitly known in the partial path. The symbolic execution state is kept in an incremental form, so that additions reflecting newly expanded subpaths can be easily erased if the subpath selection is revoked due to backup.

It is possible that it cannot be determined whether or not a particular subpath induces an inconsistent state. This occurs when a metaprogram variable is referenced in a metaprogram expression corresponding to part of a newly expanded subpath, where that variable has never been assigned a value. If the prefix of the newly expanded subpath is complete, this is determinate: it is an error condition sufficient to reject the subpath. If the prefix contains unexpanded invocation nodes, however, it may be possible to expand one of those nodes to establish the required precondition. In this case, a set/used analysis of SEMANOL global variables may be used to establish new path coverage constraint expressions to be applied to those particular invocation nodes which can reach nodes which set the used variable. This set/used analysis would be a static determination, and might be as simple as marking each node in which a variable appears on the left (or right) hand side of an assignment.

Observations On The Analyzer

The foregoing presentation has generally ignored the question of how input data might be generated for test programs. In the SEMANOL metaprogram, information that is to be read (i.e., that is not part of the program text itself) is associated with the SEMANOL #INPUT variable. In the path generation phase of the Analyzer, path conditions may then be generated that impose requirements (i.e., constraints) upon the #INPUT string. The question is how best to construct an input string that meets these constraints.

One option is to collect the #INPUT-related constraints, which may include expressions dealing with both the input string and the program, and to present these to the user for solution. This appears to be a reasonable approach, as the #INPUT-related constraints should be rather simple for test programs created by the Analyzer. Thus a user can easily produce a satisfactory input string, whereas automated methods might need to labor long to devise a solution. In the case of purely numeric input values, linear programming or conjugate gradient algorithms as used in [Clarke 1976] and [Boyer et al 1975] are candidate automation methods. However, the inclusion of any such mechanism in the Analyzer is clearly a major task, and one that was felt inappropriate to consider at this stage of development.

Now it is possible for the Synthesizer to be conveniently used in generating the input string, provided the SEMANOL metaprogram defining the language under test is written so as to include a grammar for the #INPUT variable (as TRW's specification of BASIC does). The syntactic structure of the input string is thus expressed in a manner that is directly usable by the Synthesizer, and the constraints imposed on the input in order to insure legitimate paths are likewise formulated in a convenient form for the Synthesizer. Thus proper organization of the SEMANOL metaprogram can largely make input data generation solvable by the methods already developed; it need not be treated as a separate design problem.

We feel this can often be a good way to approach this problem.

We were also prompted to consider input generation in a limited way in this project because some programming languages, such as JOVIAL (J73), lack input operators to be tested; thus other factors warranted greater attention. At the present time, it would seem most reasonable to leave input generation to the user, with Analyzer support, and worry about automation of this aspect after a prototype Analyzer implementation has succeeded.

It has been said before in this report that the user will need to collaborate with the Analyzer. The reason for this is that some problems are easily solved by a user, but are very hard to solve, or perhaps cannot be solved at all, by computer programs; thus user assistance tends to create a workable system, and to simplify implementation so that a prototype system can be developed with a reasonable effort. Unsolved problems for the designer necessarily are left for the user, regardless of difficulty. It is thought that SEMANOL metaprogram subsetting (or partitioning) provides a convenient way in which to introduce the user interface. This approach does depend upon the modularity of the SEMANOL metaprogram, in particular, for its effective operation, upon the limited communication channels that exist between modules, and upon the use of simple control flow and data structures. The modules defining control flow semantics, evaluation semantics, and storage modeling are obvious candidates well qualified for the subsetting. The simple control flow and data structures simplify consistency checking when only internal properties are involved, while the limited communication channels (i.e., global variables and parameters) simplify the user interface.

Several changes to the path generation algorithm are needed to support this approach. The notion of a complete path must be revised to consider

paths complete when they include invocation nodes for modules not in the relevant subset; such nodes cannot be expanded further. Symbolic expression values then must allow for such unexpanded definition calls. Consistency checking must be modified to distinguish between indeterminate cases that depend upon properties resolvable within the subset and those resolvable only with user assistance; in the latter situation, user messages are generated. As suggested earlier, these modifications to the basic algorithm appear suitable for any early Analyzer implementation.

4.3 Test Effectiveness Measurement

Since a SEMANOL specification is a program, it is reasonable to consider applying the same test effectiveness measures to it that are currently proposed for programs generally. In particular, the ideas of measuring test effectiveness in terms of statement coverage, branches taken, and paths executed were evaluated in relation to SEMANOL. While SEMANOL is indeed a programming language, it is a rather special one whose semantic #DF's resemble LISP (i.e., McCarthy) conditionals and whose syntactic #DF's are similar to BNF notation; this special nature is reflected in the choice of structural element definitions. For syntactic #DF's, it seems natural to consider each case of each production as a structural element. For the other statement types, the choice depends upon the test measure involved and the measurement method to be implemented.

Several options were studied. One would deal with semantic #DF's and #PROC-DF's as structural elements and measure effectiveness in terms of element coverage. This method would require that the Translator generate SIL code, of the form element-name/ELMT MARK/OP, ahead of the code generated for each structural element. The Executer would then be augmented to process the MARK/OP by flagging the associated element as having been executed. Upon conclusion of an Executer run, a list of those elements executed and those not executed could be generated. A refinement of this method could use a finer definition of structural elements such that each case of a semantic #DF and each statement of a #PROC-DF would be defined as a structural element. The implementation of this more detailed method would be much the same as already explained, but it would provide a form of test measurement in which one would have greater confidence.

AD-A052 726 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
AUTOMATED COMPILER TEST CASE GENERATION.(U)
FEB 78 P T BERNING, E R ANDERSON, F C BELZ F30602-76-C-0255
RADC-TR-78-30 NL

UNCLASSIFIED

2 OF 2
AD
A052 726

END
DATE
FILMED
5-78
DDC

A yet more sophisticated form of test effectiveness measure studied deals with marking conditional branch choices made; that is, test effectiveness is determined by whether the various control options at each branch point are taken with a given input set (e.g., that both the true and false paths are executed for a simple IF statement). The code that the Translator must generate here is similar to that shown previously, except that it must precede and follow each conditional branch point. In this case, the Executer will interpret the MARK/OP differently and will count the number of times that it has been executed. Analysis of these counts then allows the Executer to determine which branches have been taken and which have not.

These methods all appear to provide useful measures, of increasing validity, that can be added to the SEMANOL Interpreter in a straightforward manner. They are easily understood, since they are stated in terms of the SEMANOL specification itself, and so can be effectively incorporated into iterative procedures that involve analysis of test effectiveness results.

It is also possible to consider applying the notion of coverage or branch analysis to the SIL code generated, rather than at the source level discussed earlier. The structural elements in this case would become the linear strings of SIL code appearing between branch points (not each SIL element itself) and a MARK/OP would be interjected into each structural element by the Translator and interpreted by the Executer as explained. The implementation of this method in the SEMANOL Interpreter presents no special problems, but the programmer analyzing this form of test measurement output will have a more difficult task since the data is expressed in terms of generated SIL code and not the source SEMANOL code. Despite this, the method is thought to be worth further consideration.

The case of syntax #DF's does not present the problems that exist with the semantic part of a SEMANOL description. We presently envision that test effectiveness for the grammar will be measured simply in terms of syntactic case coverage. This can be accomplished without change to the Translator (i.e., without change to code generation) and with the Executer modifications essentially being restricted to the #CONTEXT-FREE-PARSE-TREE operator. As a program is parsed, the cases of syntactic #DF's that appear in the resulting parse tree are recorded and later output. While this form of measurement is not totally satisfying, it appears adequate for an automated procedure.

As can be seen, test effectiveness measurement was investigated in considerable detail; various methods were studied and implementation techniques thoroughly determined. This aspect of automated compiler test case generation is on a sound basis.

5. CONCLUSIONS

This project has attempted to find practical ways in which the generation of compiler test cases could be increasingly automated. This search has involved extensive investigation into related topics of AI research in an effort to find techniques that might be adaptable to this specific purpose. A degree of borrowing and synthesis has taken place (few ideas are altogether new), as has a great deal of independent analysis and algorithm design. The ideas of constraint propagation and partial evaluation developed for the Synthesizer, as well as the ideas of path coverage constraints and finding paths through recursive modules, developed for the Analyzer, are new products of this project. This effort has led to an improved understanding of automatic compiler test case generation, and in that understanding being embodied in a high-level design for the computerized test generation system described in this report. This design is somewhat uneven in its level of detail, but it is already sufficient to serve as the basis for implementation of a semi-automatic system that depends heavily upon manual constraint generation. A framework for incremental implementation and experimentation has thus been developed.

We would argue that prototype implementation will be needed to determine the performance acceptability of the proposed Synthesizer, and to what degree the potential Synthesizer improvements cited in this report will be needed. Likewise, the various forms of the user interface can only be appropriately developed through user experimentation with a prototype test case generation system. The eventual quality of the test cases produced by this design can only be satisfactorily decided after test cases are actually produced. This is the type of product that cannot be adequately defined a priori; thus a prototype system, employing largely manual analysis, ought to be implemented from the design given in this report. Further research into automated analysis is also recommended in the hope that techniques supporting fuller automation can be found. Prototype implementation, together with continued research, can do much to advance software science in this important area of automated testing.

6. REFERENCES

Berliner, H. J., Some Necessary Conditions for a Master Chess Program, IJCAI-73 Advance Papers, Stanford University (August 20-23, 1973), 77-85.

Bitner, J. R. and E. M. Reingold, Backtrack Programming Techniques, CACM 18, 11 (November 1975), 651-656.

Boyer, R. S., B. Elspas, and K. N. Levitt, SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution, Proceedings of the International Conference on Reliable Software (April 1975), 234-245.

Clarke, Lori A., A System to Generate Test Data and Symbolically Execute Programs, IEEE Software Engineering, SE-2, 3 (September 1976), 215-222.

Gabow, H. N., S. N. Makeshwari, and L. J. Osterweil, On Two Problems in the Generation of Program Test Paths, IEEE Trans. Software Engineering SE-2, 3 (September 1976), 227-231.

Gerhart, S. L. and L. Yelowitz, Control Structure Abstractions of the Backtracking Programming Technique, IEEE Trans. Software Engineering SE-2, 4 (December 1976), 285-292.

Golomb, S. W. and L. D. Baumert, Backtrack Programming, JACM 12, 4 (October 1965), 516-524.

Hanford, K. V., Automatic Generation of Test Cases, IBM Systems Journal 9, 4 (1970), 242-257.

Hearn, A. C., REDUCE 2: A System and Language for Algebraic Manipulation, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM (1971), 128-133.

Hoffman, R. H., User Information for the Interactive Automated Test Data Generator (ATDG) System, Johnson Space Center Internal Note No. 75-FM-88, Lyndon B. Johnson Space Center, Houston, Texas (January 1976).

Howden, William E., Symbolic Testing and the DISSECT Symbolic Evaluation System, IEEE Trans. Software Engineering SE-3, 4 (July 1977), 266.

Huang, J. C., An Approach to Program Testing, ACM Computing Surveys 7, 3 (September 1975), 113-128.

King, James C., Symbolic Execution and Program Testing, CACM 19, 7 (July 1976), 385-394.

Lombardi, L. A., and B. Raphael, LISP as the language for an incremental compiler, in E. Berkeley and D. Bobrow, eds., The Programming Language LISP: Its Operation and Applications, MIT Press, Cambridge, MA (1964), 204-219.

Mazlack, L. J., Computer Construction of Crossword Puzzles using Precedence Relationships, Artificial Intelligence 7, 1 (Spring 1976), 1-19.

Miller, E. F. and R. A. Melton, Automated Generation of Testcase Datasets, Proceedings of the International Conference on Reliable Software (April 1975), 51-58.

Moses, Joel, Algebraic Simplification: A Guide for the Perplexed, CACM 14, 8 (August 1971), 527-537.

Nilsson, N. J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York (1971).

Sacerdoti, E. D., A Structure for Plans and Behavior, Elsevier, New York (1977).

Stallman, R. M., and G. J. Sussman, Forward Reasoning and Dependency - Directed Backtracking in a System for Computer-Aided Circuit Analysis, MIT AI Lab, Memo No. 380, Cambridge (September 1976).

Sussman, G. J. and D. V. McDermott, Why Conniving is Better than Planning Artificial Intelligence Memo No. 255A, MIT (1972).

7. BIBLIOGRAPHY

Allen, F. E., Interprocedural data flow analysis, Information Processing 74 (August 1974), 398-402.

Anderson, E. R., and D. M. Heimbigner, SEMANOL(76) Interpreter Documentation, prepared for RADC Contract F30602-76-C-0238, TRW DSSG, Redondo Beach, CA (June 16, 1977).

Beckman, L., A. Haraldson, O. Oskarsson, and E. Sandewall, A partial Evaluator, and its use as a programming tool, Artificial Intelligence 7, 4 (Winter 1976), 319-357.

Belz, F. C., SEMANOL(76) Reference Manual, prepared for RADC Contract F30602-76-C-0238, TRW DSSG, Redondo Beach, CA (June 30, 1977).

Berlekamp, E. R., Program for double-dummy bridge problems - a new strategy for mechanical game playing, JACM 10, 3 (July 1963), 357-364.

Berliner, H. J., Some necessary conditions for a master chess program, IJCAI-73 Advance Papers, Stanford University (Aug. 20-23, 1973), 77-85.

Berning, P.T., Improvements to SEMANOL, prepared for RADC Contract F30602-76-C-0238, TRW DSSG, Redondo Beach, CA (June 4, 1977).

Berning, P.T., An Introduction to SEMANOL, TRW DSSG, Redondo Beach, CA (1977).

Birman, Alexander, and William H. Joyner, Jr., A problem-reduction approach to proving simulation between programs, IEEE Trans. Software Engineering SE-2, 2 (June 1976), 87-96.

Bitner, J. R. and E. M. Reingold, Backtrack programming techniques, CACM 18, 11 (November 1975), 651-656.

Bledsoe, W. W., Non-resolution theorem proving, Artificial Intelligence 9, 1 (August 1977), 1-35.

Bobrow, D. G. and B. Raphael, New programming languages for artificial intelligence research, Computing Surveys 6, 3 (Sept. 1974), 153-174.

Bochmann, G. V., Semantic evaluation from left to right, CACM 19, 2 (Feb. 1976), 55-62.

Boyer, R. S., B. Elspas, and K. N. Levitt, SELECT--A formal system for Testing and debugging programs by Symbolic Execution, Proceedings of the International Conference on Reliable Software (April 1975), 234-245.

Brown, F. M., Doing arithmetic without diagrams, Artificial Intelligence 8, 2 (April 1977), 175-200.

Brown, J. R., and M. Lipow, Testing for software reliability, TRW Software Series TRW-SS-75-02 (January 1975). Also in Proceedings of the International Conference on Reliable Software (April 1975), 518-527.

Buchanan, B. G., G. L. Sutherland, and E. A. Feigenbaum, Rediscovering some problems of artifical intelligence in the context of organic chemistry, Machine Intelligence 5, American Elsevier, New York (1970), 253-280.

Caine, Farber, and Gordon, Inc., Program Design Language Reference Guide (February 1977).

Chandrasekaran, B., Artifical intelligence - the past decade, in Advances in Computers, 13, Academic Press, New York (1975).

Clarke, Lori A., A system to generate test data and symbolically execute programs, IEEE Software Engineering, SE-2, 3 (September 1976), 215-222.

Culpepper, L. M., A system for reliable engineering software, Proceedings of the First International Conference on Reliable Software (April 1975), 186-192. Also in IEEE Trans. Software Engineering SE-1,2 (June 1975), 174-178.

Darlington, J., Automatic program synthesis in second-order logic, IJCAI-73 Advance Papers, Stanford University (Aug. 20-23, 1973), 537-542.

Davis, R. and J. King, An Overview of Production Systems, Computer Science Department Report No. STAN-CS-75-524, Stanford University (Oct. 1975).

deKleer, J., J. Doyle, G. L. Steele, Jr., and G. J. Sussman, Explicit control of reasoning, MIT AI Lab, AI Memo 427 (June 1977).

Deutsch, L. P., An Interactive Program Verifier, Xerox PARC CSL-73-1 (May 1973), reprinted July 1976.

Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J. (1976).

Doran, J., New developments of the graph traverser, Machine Intelligence 2, American Elsevier, New York (1968), 119-135.

Elspas, B., R. S. Boyer, R. E. Shostak, and J. M. Spitzen, A verification system for JOVIAL/J3 programs (rugged programming environment-rpe/1), Rome Air Development Center Technical Report RADC-TR-76-58 (March 1976), AD# A024 395.

Elspas, B., R. E. Shostak, and J. M. Spitzen, A verification system for JOCIT/J3 programs (rugged programming environment-rpe/2), Rome Air Development Center Technical Report RADC-TR-77-229 (June 1977), AD# A042 670.

Fairly, Richard E., An experimental program-testing facility, Proceedings of the First National Conference on Software Engineering (September 1975), 47-55. Also in IEEE Trans. Software Engineering SE-1, 4 (December 1975), 350-357.

Floyd, R. W., Nondeterministic algorithms, JACM 14, 4 (October 1967), 636-644.

Fosdick, L. D., and L. J. Osterweil, Data flow analysis in software reliability, ACM Computing Surveys 8, 3 (September 1976), 305-330.

Freuder, Eugene C., Synthesizing Constraint Expressions, MIT Artificial Intelligence Lab, AI Memo No. 370 (July 1976).

Gabow, H. N., S. N. Makeshwari, and L. J. Osterweil, On two problems in the generation of program test paths, IEEE Trans. Software Engineering SE-2, 3 (September 1976), 227-231.

Gerhart, S. L., and L. Yelowitz, Observations of fallibility in applications of modern programming methodologies, IEEE Trans. Software Engineering SE-2, 3 (September 1976), 195-207.

Gerhart, S. L. and L. Yelowitz, Control structure abstractions of the backtracking programming technique, IEEE Trans. Software Engineering SE-2, 4 (December 1976), 285-292.

Goldstein, I. P., Bargaining between goals, IJCAI-75 Advance Papers 1, Tbilisi, Georgia, USSR (September 3-8, 1975), 175-180.

Goldstein, I. P. and M. L. Miller, Structured planning and debugging - A Linguistic Approach to Problem solving, AI Working Paper 125, MIT Artificial Intelligence Lab (June 8, 1976).

Golomb, S. W. and L. D. Baumert, Backtrack programming, JACM 12, 4 (October 1965), 516-524.

Goodenough, J. B. and S. L. Gerhart, Toward a theory of test data selection, Proceedings of the International Conference on Reliable Software (April 1975), 493-510.

Green, C., Application of theorem proving to problem solving, IJCAI-69 Proceedings, Washington, D. C. (May 7-9, 1969), 219-239.

Griswold, Ralph E., Language facilities for programmable backtracking, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, appearing in SIGPLAN Notices 12, 8 (August 1977), 94-99.

Hall, Andrew D., Jr., The ALTRAN system for rational function manipulation--a survey, CACM 14, 8 (August 1971), 517-521.

Hamlet, Richard G., Testing programs with the aid of a compiler, IEEE Trans. Software Engineering SE-3, 4 (July 1977), 279-290.

Hanford, K. V., Automatic generation of test cases, IBM Systems Journal 9, 4 (1970), 242-257.

Harris, L. R., The heuristic search under conditions of error, Artificial Intelligence 5, 3 (Fall 1974), 217-234.

Harrison, William H., Compiler analysis of the value ranges for variables, IEEE Trans. Software Engineering SE-3, 3 (May 1977), 243-250.

Hayes-Roth, F., and D. J. Mostow, An automatically compilable recognition network for structured patterns, IJCAI-75 Advance Papers 1 (September 3-8, 1975), 246-251.

Hearn, A. C., **REDUCE** 2: A system and language for algebraic manipulation, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM (1971) 128-133.

Henderson, P., Finite state modelling in program development, Proceedings of the International Conference on Reliable Software (April 1975), 221-227.

Hetzel, W. C. (Ed.), Program Test Methods, Prentice-Hall, Englewood Cliffs, N.J. (1973).

Hewitt, C., P. Bishop and R. Steiger, A universal modular ACTOR formalism for artificial intelligence, IJCAI-73 Advance Papers, Stanford University (Aug. 20-23, 1973), 235-245.

Hillier, F. S., and G. J. Lieberman, Operations Research, Holden-Day, Inc., San Francisco (1974).

Hoffman, R. H., User information for the interactive automated test data generator (ATDG) system, Johnson Space Center Internal Note No. 75-FM-88, Lyndon B. Johnson Space Center, Houston, Texas (January 1976).

Holberton, F. E., and E. G. Parker, NBS FORTRAN Test Programs, Vol. 1-3, National Bureau of Standards Special Publication 399 (October 1974).

Hollerbach, J. M., Hierarchical Shape Description of Objects by Selection and Modification of Prototypes, MIT AI Laboratory, AI-TR-346, Cambridge, Mass. (November 1975).

Hopcroft, John, and Robert Tarjan, A447-efficient algorithms for graph manipulation, CACM 16, 6 (June 1973), 372-378.

Horowitz, E., and S. Sahni, Algorithms: Design and Analysis, Computer Science Press, Potomac, Maryland (1977).

Howden, William E., Experiments with a symbolic evaluation system, Proceedings of the National Computer Conference (1976), 899-908.

Howden, W. E., Reliability of the path analysis testing strategy, IEEE Trans. Software Engineering SE-2, 3 (September 1976), 208-214.

Howden, William E., Symbolic testing and the DISSECT symbolic evaluation system, IEEE Trans. Software Engineering SE-3, 4 (July 1977), 266.

Huang, J. C., An approach to program testing, ACM Computing Surveys 7, 3 (September 1975), 113-128.

Ikezawa, Michael A., and Richard E. Kayfes, A structural calculus for program analysis and testing, Logicon Report No. CSS-75019, presented at the 9th Annual Asilomar Conference on Circuits, Systems, and Computers (November 1975).

Jackson, P. C., Jr., Introduction to Artificial Intelligence, Petrocelli Books, New York (1974).

Kibler, D. F., J. M. Neighbors, and T. A. Standish, Program manipulation via an efficient production system, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, appearing in SIGPLAN Notices 12, 8 (August 1977).

King, James C., A new approach to program testing, Proceedings of the International Conference on Reliable Software (April 1975), 228-233.

King, James C., Symbolic execution and program testing, CACM 19, 7 (July 1976), 385-394.

Knuth, D. E., Deletions that preserve randomness, IEEE Trans. Software Engineering SE-3, 5 (September 1977), 351-358.

Krause, K. W., R. W. Smith, and M. A. Goodwin, Optimal software test planning through automated network analysis, TRW Software Series TRW-SS-73-01 (April 1973). Also in the Proceedings of IEEE Computer Software Reliability Symposium (April 1973).

Ledgard, H. F., Production systems: or can we do better than BNF? CACM 17, 2 (February 1974), 94-102.

Lee, R. C. T., C. L. Chang, and R. J. Waldinger, An improved program synthesizing algorithm and its correctness, CACM 17, 4 (April 1974), 211-217.

Levi, G., and F. Sirovich, A problem reduction model for non-independent subproblems, IJCAI-75 Advance Papers 1, Thilisi, Georgia, USSR (September 3-8, 1975), 340-344.

Levi, G. and F. Sirovich, Generalized and/or graphs, Artificial Intelligence 7, 3 (Fall 1976), 243-259.

Lewis, P. M. II, D. J. Rosenkrantz, and R. E. Stearns, Compiler Design Theory, Addison-Wesley Publishing Company, Inc., Reading, MA and Menlo Park, CA (1976).

Lipow, M. Applications of algebraic methods to computer program analysis, TRW Software Series TRW-SS-73-10 (May 1973).

Lombardi, L. A. and B. Raphael, LISP as the language for an incremental compiler, in E. Berkeley and D. Bobrow, eds., The Programming Language LISP: Its Operation and Applications, MIT Press, Cambridge, MA (1964), 204-219.

McDermott, D., SYMBOL-MAPPING: A technical problem in PLANNER-like systems, SIGART Newsletter 51 (April 1975), 4-5.

McDermott, D., Artificial intelligence meets natural stupidity, SIGART Newsletter 57 (April 1976), 4-9.

Mackworth, A. K., Consistency in networks of relations, Artificial Intelligence 8, 1 (February 1977), 99-118.

Manna, Z. and R. Waldinger, Knowledge and reasoning in program synthesis, Artificial Intelligence 6, 2 (Summer 1975), 175-208.

Marsh, D. L., Memo functions, the graph traverser, and a simple control situation, in Meltzer, B. and D. Michie (eds.), Machine Intelligence 5, American Elsevier, New York (1970), 281-300.

Mazlack, L. J., Computer construction of crossword puzzles using precedence relationships, Artificial Intelligence 7, 1 (Spring 1976), 1-19.

Michie, D., Strategy-building with the graph traverser, in Collins, N.L., and D. Michie (Eds.), Machine Intelligence 1, American Elsevier, New York (1967), 135-152.

Miller, E. F. and R. A. Melton, Automated generation of testcase datasets, Proceedings of the International Conference on Reliable Software (April 1975), 51-58.

Miller, E. F., Methodology for comprehensive software testing, Rome Air Development Center Technical Report RADC-TR-75-161 (June 1975). Also available as U. S. Dept. of Commerce NTIS document AD-A013111.

Miller, W. and D. L. Spooner, Automatic generation of floating-point test data, IEEE Trans. Software Engineering SE-2, 3 (September 1976), 223-226.

Moses, Joel, Algebraic simplification: a guide for the perplexed, CACM 14, 8 (August 1971), 527-537.

Motto, R. M., et al, JOVIAL compiler implementation tool (JOCIT) test report on the HIS 6080, RADC Internal Memo (August 1974).

Nevins, A. J., Plane geometry theorem proving using forward chaining, Artificial Intelligence 6, 1 (Spring 1975), 1-23.

Nevins, A. J., A relaxation approach to splitting in an automatic theorem prover, Artificial Intelligence 6, 1 (Spring 1975), 25-39.

Newell, A., Some problems of basic organization in problem-solving programs, in Self Organizing Systems, Pergamon, New York (1962), 393-423.

Nilsson, N.J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York (1971).

Osterweil, Leon J. and Lloyd D. Fosdick, Some experience with DAVE--a FORTRAN program analyzer, Proceedings of the National Computer Conference (1976), 909-915.

Paige, M. R., and E. E. Balkovich, A test plan for a structured program, General Research Corporation Releasable Memorandum RM-1658/1 (October 1972).

Paige, Michael R., Program graphs, an algebra, and their implication for programming, IEEE Trans. Software Engineering SE-1, 3 (September 1975), 286-291.

Proceedings of the International Conference on Reliable Software (April 1975), also reprinted in SIGPLAN Notices 10, 6 (June 1975) and excerpted in IEEE Trans. Software Engineering SE-1, 2 (June 1975).

Quinlan, J. R., A task-independent experience-gathering scheme for a problem-solver, IJCAI-69 Conference Proceedings, Washington, D. C. (May 7-9, 1969), 193-197.

Rader, G. M., A method for composing simple traditional music by computer, CACM 17, 11 (November 1974), 631-638.

Ramamoorthy, C. V., and S. F. Ho, Testing large software systems with automated software evaluation systems, Proceedings of the International Conference on Reliable Software (April 1975), 382-394. Also in IEEE Trans. Software Engineering SE-1, 1 (March 1975), 46-58.

Ramani, S. and A. Newell, On the Generation of Problems, Department of Computer Science, Carnegie-Mellon University (November 1973).

Rosen, Barry K., High level data flow analysis, part 1 (Classical structured programming), IBM Research Report RC 5598 (#24200), (August 1975).

Rosen, Barry K., Data flow analysis for procedural languages, IBM Research Report RC 5948 (#25807), (April 1976).

Rulifson, J. F., J. A. Derksen, and R. J. Waldinger, QA4: A Procedural Calculus for Intuitive Reasoning, AI Center Technical Note 73, SRI Project 8721 (November 1972).

Sacerdoti, E. D., Planning in a hierarchy of abstraction spaces, Artificial Intelligence 5, 2 (Summer 1974), 115-135.

Sacerdoti, E., The nonlinear nature of plans, IJCAI-75 Advance Papers, Tbilisi, Georgia, USSR (September 1975), 206-218.

Sacerdoti, E. D., A Structure for Plans and Behavior, Elsevier, New York (1977).

Sager, N. and R. Grishman, The restriction language for computer grammars of natural language, CACM 18, 7 (July 1975), 390-400.

Samet, H., Automatically Proving the Correctness of Translations Involving Optimized Code, Stanford AI Lab Memo A-IM-259 (May 1975).

Samet, H., A normal form of compiler testing, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, appearing in SIGPLAN Notices 12, 8 (August 1977), 155-162.

Samet, H., A machine description facility for compiler testing, IEEE Trans. Software Engineering SE-3, 5 (September 1977), 343-350.

Sandewall, E. J., Concepts and methods for heuristic search, IJCAI-69 Conference Proceedings, Washington, D. C. (May 7-9, 1969), 199-218.

Shannon, C., Programming a digital computer for playing chess, Philosophy Magazine 41 (March 1950), 356-375. Reprinted in Newman, J. R., (ed.), The World of Mathematics, Vol. 4, Simon and Schuster, New York (1954).

Shen, S. N. T., and E. R. Jones, A number theory approach to problem representation and solution, IJCAI-73 Advance Papers, Stanford University (August 20-23, 1973), 606-611.

Shimura, M., and F. H. George, Rule-oriented methods in problem solving, Artificial Intelligence 4, 3-4 (Winter 1973), 203-223.

Shortlittle, E. H., Computer Based Medical Consultations: MYCIN, American Elsevier, New York (1976).

Simon, H. A., Experiments with a heuristic compiler, JACM 10, 4 (October 1963), 493-506.

Stallman, R. M., and G. J. Sussman, Forward Reasoning and Dependency - Directed Backtracking in a System for Computer-Aided Circuit Analysis, MIT AI Lab, Memo No. 380, Cambridge (September 1976).

Standish, T., D. C. Harriman, D. F. Kibler, and J. M. Neighbors, The Irving program transformation catalog, Computer Science Department, University of California at Irvine (January 1976).

Summers, P. D., A methodology for LISP program construction from examples, JACM 24, 1 (January 1977), 161-175.

Sussman, G. J., A Computational Model of Skill Acquisition, AI TR-297, Artificial Intelligence Lab, MIT, Cambridge, Mass. (August 1973).

Sussman, G. J., Electric design a problem for artificial intelligence research, IJCAI-77 Conference Proceedings, Cambridge, MA (August 22-25, 1977), 894-900.

Sussman, G. J. and D. V. McDermott, Why conniving is better than planning, Artificial Intelligence Memo No. 255A, MIT (1972).

Tate, Austin, Interacting goals and their use, IJCAI-75, Advance Papers (September 1975), 215-218.

VanderBrug, G. J., and J. Minker, State-space, problem-reduction, and theorem proving - some relationships, CACM 18, 2 (February 1975), 107-115.

Wilks, Y., An intelligent analyzer and understander of English, CACM 18, 5 (May 1975), 264-274.

Winograd, T., Procedures as a representation for data in a computer program for understanding natural language, MAC TR-84, MIT Project MAC (February 1971).

Winston, P.H., New progress in artifical intelligence, MIT AI Lab, AI-TR-310, Cambridge, Mass. (1974).

Winston, P.H., (Ed.), The Psychology of Computer Vision, McGraw Hill, New York (1975).

Winston, P. H. Artificial Intelligence, Addison-Wesley, Reading, MA (1977).

Zislis, Paul M., Semantic decomposition of computer programs: an aid to program testing, Purdue University Computer Science Department Technical Report 110 (January 1974). Also was to appear in Acta Informatica; exact reference unknown.

Zobrist, A. L., and F. R. Carlson, Jr., Detection of combined occurrences, CACM 20, 1 (January 1977), 32-35.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

AMERICAN REVOLUTION BICENTENNIAL 1776-1976